NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AN INVESTIGATION OF SEVERAL BRANCHING FUNCTIONS IN A BRANCH AND BOUND ALGORITHM FOR THE CHROMATIC NUMBER PROBLEM

by

Ronald Ernest Rautenberg

December, 1980

Thesis Advisor: Douglas R. Smith

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
An Investigation of Several Branching Functions in a Branch and Bound Algorithm for the chromatic number Problem

5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis; Dec 80

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Ronald Ernest Rautenberg

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93940

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93940

12. REPORT DATE
December, 1980

13. NUMBER OF PAGES
124

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)
Naval Postgraduate School
Monterey, California 93940

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
chromatic number, graph coloring, k-chromatic,
branch and bound, vertex ordering, articulation points

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The chromatic number problem is to determine the minimum number of colors to assign to the vertices of a graph such that no connected vertices are assigned the same color. This paper presents a branch and bound solution to the chromatic number problem and investigates five different branching functions. Additionally, a method of coloring very sparse graphs is presented which divides a graph into biconnected components and reduces the time required to color the graph.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Approved for public release; distribution unlimited.

An Investigation of Several Branching Functions in a Branch and Bound Algorithm for the Chromatic Number Problem

by

Ronald E. Rautenberg
Lieutenant Commander, United States Navy
B.S., University of Washington, 1970

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1980

ABSTRACT

The chromatic number problem is to determine the minimum number of colors to assign to the vertices of a graph such that no connected vertices are assigned the same color. This paper presents a branch and bound algorithm for the solution to the chromatic number problem and investigates five different branching functions. Additionally, a method of coloring very sparse graphs is presented which divides a graph into biconnected components and reduces the time required to color the graph.

TABLE OF CONTENTS

I. INTRODUCTION 7

II. BACKGROUND 11

A. GRAPH THEORY 11

B. BRANCH AND BOUND 14

C. PROBLEM COMPLEXITY 19

III. THE ALGORITHMS 27

A. GRAPH GENERATION 27

B. GRAPH DIVIDING 32

C. GRAPH COLORING 38

D. THE VERTEX ORDERING FUNCTIONS 45

1. Random 47

2. Degree 47

3. Dvec 3 48

4. Adjacency 49

5. Least Colors Available 50

E. CONTROLLING ALGORITHM..... 52

F. GENERAL COMMENTS ON ALGORITHM DEVELOPMENT AND THE 'C' PROGRAMMING LANGUAGE 54

IV. RESULTS 56

A. PARAMETERS 56

1. Time 57

2. Space 63

B. TIME ANALYSIS 64

C. SPACE ANALYSIS 68

D. ADVANTAGE OF DIVIDING THE GRAPHS 72

V. CONCLUSIONS 77

APPENDIX A--PROGRAM SOURCE CODE 80

APPENDIX B--TABULATION OF RESULTS 102

LIST OF REFERENCES 122

INITIAL DISTRIBUTION LIST..... 124

LIST OF FIGURES

1. The Tree Structure for the Colorability Problem. . . 17

2. Comparison of Several Polynomial and Exponential Complexity Functions 21

3. Transformation of Problem Q to Problem R 23

4. Examples of the Isomorphs of 2 Distinct Graphs . . . 29

5. A Graph with Articulation Points 32

6. A Graph Divided at its Articulation Points 33

7. A Depth First search of a Graph 35

8. A Schematic of a Graph with 3 Biconnected Components and 2 Articulation Points 36

9. Two Paths P and Q which are not Isomorphic 44

10. Example of Shortcoming of Degree Method 48

11. The First Four Levels of the Unpruned Search Tree. . 60

12. Average Expansions vs. Graph Size 65

13. Maximum Expansions vs. Graph Size 66

14. Average Nodes vs. Graph Size 69

15. Maximum Nodes vs. Graph Size 70

16. Average Branching Factor vs. Depth in the Tree . . . 71

17. Ratio of Largest Biconnected Component to the Original Graph Size 73

18. Ratio of Expansions Required for Divided Graph to Expansions Required for Undivided Graph 75

I. INTRODUCTION

Informally, a graph is a collection of points, called nodes or vertices, some of which may be connected by lines, called edges. A solution to the graph colorability problem specifies a color for each node such that any two nodes which are connected by an edge do not have the same color. The optimal colorability problem is to assign colors, as in the graph colorability problem, with the restriction that the fewest number of colors possible is used.

For centuries, mathematicians have studied graphs and their characteristics with the awareness that many engineering and mathematical problems could be modeled by a graph. Then questions about the engineering problem could be answered by studying the characteristics of the graph and using knowledge derived from graph theory.

One of the earliest and best documented graph colorability problems dates back to 1852, when a student of De Morgan named Francis Guthrie, proposed that 4 colors was all that was needed to color a map such that no two bordering countries are the same color[Ref. 1]. Note that this problem is transposed into a graph colorability problem by using a node for each country, and connecting 2 nodes with an edge if the two corresponding countries share a common border.

In fact, beginning in the 19th century, many of the engineering problems in areas such as scheduling, communications, transportation, electronics, and chemistry have been studied through the use of graph theory. More recently, the social, biological, and environmental sciences have relied on graph theory to investigate specific problems. In the decade of the seventies, computer science and graph theory have been closely intertwined as computer scientists realized that many of their difficult problems can best be understood by graph modeling. One of the most notable is the scheduling problem where two or more processes must compete for common resources. Ways to efficiently schedule these processes are better understood when the system is modeled by a graph.

The intent of this research has been to take a single problem from graph theory, namely optimal graph colorability, and to investigate the use of the digital computer to find optimal colorings for various graphs. Though a seemingly trivial problem on a computer, it is complicated by the fact that in the worst case, all known methods of finding solutions to problems of this type require exponential time. That is, as the size of the graph gets larger, the time required to find the optimal coloring grows exponentially.

Typically, the researcher investigating an exponential time problem, whose size is very large, is faced with one of two choices: 1) he must accept a solution which is not necessarily optimal, but some approximate or 'close' solution, or 2) he must relax some of the restrictions on the problem. Most of the research done on the graph colorability problem has been in the area of finding approximation algorithms, that is, algorithms which attempt to find a relatively few number of colors without undue computational effort being expended[Refs. 2,3,4]. The primary thrust of this research has been to investigate methods of finding only optimal colorings, and to study the effects of different procedures on the time and space (computer memory) required by each.

Specifically, the branch and bound technique is used to find the optimal coloring of a large number of randomly generated graphs, and the branching function is varied to determine the relative value of different heuristic branching functions.

In summary, this research encompasses three related, yet diversified disciplines. From mathematics comes graph theory; from operations research comes the branch and bound techniques; and from computer science comes problem complexity. Background information is provided for these three to-

pics in the first part of this report. Following that is a discussion of the research done on the computer, including the algorithms used, the branching functions investigated, and the graph generation techniques. Finally, there is a discussion of the results obtained, the difficulties encountered, and the ideas and notions derived during the course of the research.

II. BACKGROUND

A. GRAPH THEORY

While not attempting in any way to fully cover the field of graph theory, an introduction to the notions and terminology which relate to the colorability problem is necessary for an understanding of the discussion which follows. The definitions used here are standard in most of the literature on graph theory, so the reader with some background in the field can skip this section without any loss of continuity.

Formally, a graph G = (V,E) consists of a finite, nonempty set of nodes or vertices V, and a finite, possibly empty set of vertex pairs called edges E. For the purposes of this research the set of edges is restricted to not contain any elements of the form (x,x). That is, no edge can exist from a vertex to itself. In the case where the edges have direction, they are defined by an ordered pair of vertices and the graph is called a DIGRAPH. This research has been strictly limited to non-directed graphs.

The general graph colorability problem is formalized as follows: Given graph G and integer k, does there exist a function R: V -> I where I = {1,2,...,k} such that for all (x,y) $\in$ E, R(x) $\neq$ R(y)? The optimal colorability problem is to find the smallest integer k such that a function R exists. That smallest integer found is called the chromatic

number of the graph G. If R(x) = R(y) then vertices x and y are said to belong to the same color class. Note that the function R partitions the vertices of the graph into k color classes.

Some additional terms relating to graphs are:

Adjacent: Vertex x is adjacent to vertex y if an edge exists between them, that is $(x,y) \in E$.

Adjacency matrix: A two dimensional, n x n matrix, used to describe any graph. The indices into the matrix are the vertices of the graph, and the elements are a 1 if an edge exists between the two vertices, and a 0 otherwise. For the graphs used in this research the diagonal elements are all zeroes and the matrix is reflective about the diagonal. The adjacency matrix is a very common, and easy to implement method of representing a graph in a computer.

Degree: The degree of a vertex is the number of edges incident upon it, or the number of vertices which are adjacent to it. The degree of vertex x is easily calculated from the adjacency matrix by simply summing the 1's in row x or column x. If the elements of the adjacency matrix are denoted a(i,j) and the degree of vertex x is denoted d(x) then

$$d(x) = \sum_{i} a(x,i)$$

Path: A path is said to exist from vertex x to another vertex y if one can start at vertex x and follow the edges of the graph and reach vertex y. Formally, a path exists if there exists an ordering of some of the vertices of G $\{v_1, v_2, \ldots, v_n\}$ such that $v_1 = x$, $v_n = y$ and for $1 <= i <= n-1$, $(v_i, v_{i+1}) \in E$.

Connected Graph: A graph is connected if at least one path exists from each vertex to every other vertex in V.

Doubly connected graph: A graph is doubly connected if for every vertex x and every vertex y at least 2 paths exist from x to y and no other vertices are common to both paths. This is also called a biconnected graph.

Articulation point: A vertex v is an articulation point of graph G if there exists some vertex x and some vertex y such that every path from x to y includes vertex v. Note that a doubly connected graph has no articulation points.

Complete graph: A graph G is complete if all possible edges are present. That is G is complete if for all vertices x and y, $(x,y) \in E$.

Clique: A clique is a subset of a graph such that every vertex in the clique is adjacent to every other vertex in the clique.

B. BRANCH AND BOUND

Many problems which deal with searching for a solution or set of solutions satisfying some constraints can be solved using the branch and bound technique. It is especially useful in solving minimization (or maximization) problems and has been successfully used in constrained optimization problems since the late 1950's. Although there are several ways of describing branch and bound, the basic idea is to split the solution space (branch) and place a limit, or lower bound, on the optimal cost of the problem limited to each subset of the solution space. Those subsets whose optimal costs do not exceed the cost of some known solution are then repeatedly divided and bounded until a solution is found whose actual cost is no greater than the lower bound on any of the subsets. This solution is thus the optimal one. The power of branch and bound comes from its ability to leave unexplored those subsets which are known not to contain the optimal solution.

In order to use the branch and bound technique the problem must have an expressable set of solutions and a cost function COST() which maps the solutions into nonnegative integers. Usually, the problem calls for finding that solution s for which COST(s) is a minimum. Sometimes all solutions may be desired whose cost is no greater than COST(s).

Many problems require that all the solutions satisfy some set of constraints which may be divided into two categories, explicit and implicit. If a solution is represented by an n-tuple $(x_1, x_2, \ldots, x_n)$, then the explicit constraints are ones which determine what values the x_i's may take on. The implicit constraints restrict the ways in which the x_i's in a solution relate to each other. All solutions which satisfy the explicit constraints are said to belong to the possible solution space. The implicit constraints then determine which solutions in the solution space satisfy the constraints of the problem. These solutions make up the 'feasible' solution space.

For the graph colorability problem, we will use the convention that a solution is expressed as an n-tuple $(x_1, x_2, \ldots, x_n)$ of integers where $1 <= x_i <= n$, and x_i is the color assigned to vertex i. The explicit constraints restrict the x_i's to integer values from 1 to n. Therefore, the possible solution space has n^n elements(n ways of picking an integer from 1 to n for each of n positions). The implicit constraint in this problem is:

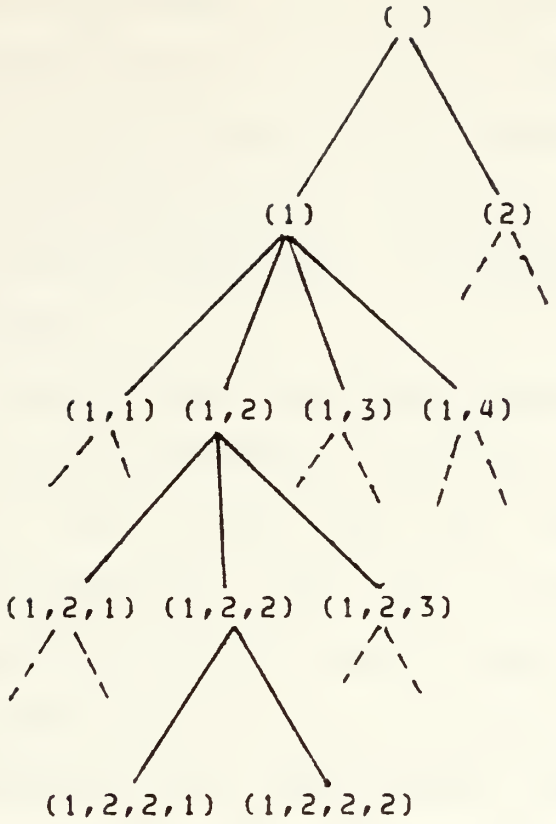
$$(i,j) \in E \Rightarrow x_i \neq x_j$$

The size of the feasible solution space is therefore also a function of the number of edges in the graph and their arrangement.

The first step of branch and bound is to systematically divide the solution space into subsets. This may be represented by a tree organization, where the root of the tree represents all feasible solutions. The children of a node N then represent the subsets into which N can be divided. The leaf, or terminal nodes are the particular solutions which make up the set of feasible solutions.

For the graph colorability problem, the following tree organization was used. The root node represents all feasible n-tuples. Then, an arbitrary vertex is chosen and the n-tuples are divided into n subsets such that subset(k) contains all the n-tuples for which $x_i = k$. This is then done repeatedly, picking a new vertex at each level of the tree. For notation purposes, a partial solution $(x_1, x_2, \ldots, x_k)$, $k < n$, is an assignment of colors to the vertices $1, 2, \ldots, k$. This k-tuple is then used to represent all the elements of a subset. Using this notation, a portion of the tree for the case $n = 4$ is shown in fig. 1. Note that the lower bound on the cost of a node is the number of different colors in the partial solution.

A branching strategy is a rule for determining how a solution is to be divided into subsets. For this problem the branching function is the method of choosing the next vertex to color at each level of the tree.



The tree structure for the Colorability Problem.
Figure 1

Once the organization is decided upon, the task of the problem solving method is to explore, or search this 'state space tree' until the optimal answer state is found. Since the time required by the search algorithm is a function of the number of nodes explored in the tree, the search strategy is to explore the fewest nodes possible. One way to facilitate this is to 'prune' the tree as early in the search as possible. Pruning takes place when a node is reached which violates the implicit constraints of the prob-

lem. Since no legal answer state could possibly be reached through that node, none of its children need be explored. The size of the search space is thus reduced by the number of nodes below the pruned node. Obviously, the higher in the tree that pruning takes place, the greater the reduction in the search space.

The second way to reduce the search space is the essence of the branch and bound method. Each node has a cost associated with it, and since the cost function is non-decreasing, no searching need take place below any node whose cost is greater than the cost of some known solution. A 'search strategy' is a rule for choosing which unexplored node to explore next in the tree search. Several different basic search strategies exist for exploring a tree.

A depth first search proceeds down the tree from the root, exploring the leftmost children of each node first. At any point where pruning takes place, the search is backed up to the first node where it can again proceed down the tree in a different path. Top-to-bottom, left-to-right is a good description of this method.

A breadth first search explores all the children of a node before proceeding down the tree. This might be called a left-to-right, top-to-bottom search.

A 'best-first' or 'least cost' search strategy explores only the most promising path through the tree at any given time. A node is explored by assigning costs to all of its children and placing them on an unexplored list. The next node to be explored is the one on the unexplored list whose cost is currently the lowest. A priority queue can be used to maintain an ordering of the unexplored nodes and at the point where the highest priority node is an answer state then the search can be terminated because that node represents the best or optimal solution to the problem. Nodes which violate the problem constraints may be assigned an arbitrarily high cost, or merely not placed in the queue, which effectively prunes the tree. This is the search strategy employed in this research effort.

This has been a general description of the branch and bound technique. More specific details of the tree arrangement, the search strategy, and the cost function used in this research will be described in Section III.

C. PROBLEM COMPLEXITY

The complexity of a problem is said to be polynomial-time if an algorithm exists for which the number of elementary or fundamental operations is limited by a polynomial of the length of an encoding of the problem. A problem is polynomial-space if an algorithm exists for which the amount

of elementary computer storage space needed for computation is never greater than some polynomial function of the encoding of the problem. An algorithm for a polynomial-time problem is said to be a 'good' algorithm and all problems for which a good algorithm exist belong together in a class called P. It may seem somewhat extravagant to group all polynomial-time algorithms into one class since polynomials can be quite large. However, no matter what the coefficients are, in the limit as n gets large, every exponential function dominates every polynomial function. Further, though unexplained, experience has shown that for many of the problems encountered which have a polynomial-time algorithm, the solution is bounded by a polynomial of small degree. Some examples are ordered searching which is O(ln n), sorting which is O(n*ln(n)), and matrix multiplication, which is $O(n^{2.81})$.

Another group of problems are those whose best 'known' algorithm require greater than polynomial time. Some of these are: 1) Algorithms requiring subexponential time, such as $O(e^{\sqrt{n}})$, 2) algorithms requiring exponential time, such as $O(e^{n})$, and 3) algorithms requiring super-exponential time, such as $O(e^{e^{n}})$. The disadvantage of non-polynomial-time algorithms is in the explosive growth of the maximum solution time as illustrated by figure 2. In this table the maximum

computing time is shown for different complexity functions as n goes from 10 to 60.

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 cent.	2×10^8 cent.	1.3×10^{13} cent.

Comparison of several polynomial and exponential time complexity functions[Ref.5]
Figure 2

Note that the only concern is with known algorithms and worst-case situations. No conclusions are made about expected or average performance of an algorithm.

Problems for which there exists an algorithm which can guess a solution and verify its correctness in polynomial-time make up the class of non-deterministic polynomial-time problems, NP. For example, graph coloring is in NP because

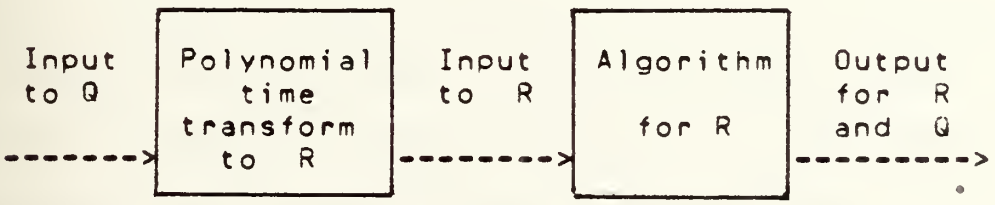
an algorithm exists which can guess a coloring and check whether or not it is legal in polynomial-time. Such an algorithm needs O(n) time to arbitrarily assign a color to each of the n vertices. Then it needs to consider each edge of the graph and check whether the 2 vertices have the same color. This can be done in constant time for each edge and since there may be $O(n^2)$ edges, the entire verification takes $O(n^2)$.

Note in all of this discussion the word 'known'. There is not one single problem in the class NP for which it has been proved that the problem is not in P[Ref. 6]. There are no polynomial-bounded algorithms known for many problems in NP, but no lower bound which is greater than polynomial time has been proven for any of these problems. Thus, it is still an open question as to whether the class NP properly contains P or whether the two classes are equal.

There is obviously much interest in trying to prove whether a given problem Q could or could not be solved in polynomial-time. For if one can prove that no polynomial-time algorithm for Q could possibly exist then there is no point in expending the effort to find such an algorithm. Short of proving that a problem has no polynomial-time algorithm, however, there is some comfort in knowing that ones difficult problem is somehow related or at least as diffi-

cult as many other problems for which no one has found a polynomial-time algorithm. For this discussion some more definitions are needed. A 'problem' is some general question to be answered which usually has one or more unspecified parameters and some specified properties which the answer is required to satisfy. An 'instance' of a problem is obtained when the parameters are specified. For example, an instance of the colorability problem is obtained by specifying the vertices and edges of the graph and the number of colors for which a coloring is desired.

A problem Q is said to be 'reducible' or 'transformable' to R if an instance of Q can be transformed in polynomial time to an instance of R. In other words, if the answer to problem Q is 'yes' if and only if the answer to R is 'yes' and if the input parameters to problem R can be determined in polynomial-time given the input parameters to Q, then Q is reducible to R. The following schematic illustrates this definition:



Transformation of Problem Q to Problem R
Figure 3

Clearly, if Q is reducible to R and R is in the class P, then Q is also in P. Note that nothing is said about the existence of an algorithm for R, only that 'if' an algorithm for R exists which runs in polynomial-time, then one also exists for Q.

If every problem in the class NP is reducible to some problem Q then Q is said to be NP-hard. In other words, Q is at least as hard as every problem in NP. Further, if Q is also in the class NP then Q is NP-complete. The significance of a problem being NP-complete is that if a polynomial-time algorithm is ever found for any NP-complete problem then it will be known that a polynomial-time algorithm exists for all NP problems and that indeed NP = P. Further, if any problem in NP is ever shown to require super-polynomial time complexity then every NP-complete problem must also require greater than polynomial time.

To show that a particular problem Q is NP-complete, then, one must show that it is indeed in NP and then either show that every NP problem could be reduced to Q or that some already proven NP-complete problem is reducible to Q. In 1971, Steven Cook[Ref. 7] laid the ground work for the current theory of NP-completeness in a paper entitled "The Complexity of theorem proving procedures." One of his most significant results was the proof that the 'Satisfiability'

problem from boolean logic is NP-complete, and it holds the distinction of being the first NP-complete problem. Reference 5 contains an excellent description of the Satisfiability problem as well as a proof of Cook's theorem. Cook also proved that a variation of the Satisfiability problem, called 3-Satisfiability, was NP-complete by transforming Satisfiability into 3-Satisfiability. Then, in 1972, Karp[Ref. 8] proved that many decision problems were also NP-complete. One of these proofs was that the 3-Satisfiability problem was reducible to the colorability problem. Thus the colorability problem is also NP-complete.

Inasmuch as most theorists are in agreement that the NP-complete problems are probably super-polynomial, much work has been done in recent years to find relaxations to the problems which allow them to be solved in polynomial time. As the graph colorability problem has many applications in real-life problems, many methods of finding approximate solutions have been tried. In many cases where a graph can be used to model a real-life problem, the absolute minimum number of colors may not be needed, but rather, some close approximation might be satisfactory. Unfortunately, even this has proved to be a difficult problem. Many algorithms have been devised[Refs. 2,3,4] to find approximate colorings, but for every one, it is possible to construct a

graph such that the approximation algorithm will find a coloring which uses many more colors than the chromatic number. Though the algorithm may indeed run in polynomial time, the closeness of the approximation to the actual chromatic number of the graph may be suspect. In fact, in 1976, Garey and Johnson[Ref. 14] proved the following:

Given graph G with chromatic number X(G), and polynomial-time approximation algorithm A, which computes a coloring of G using A(G) colors, there can be no A such that for all G:

$$\frac{A(G)}{X(G)} < r \quad \text{for } r < 2.$$

In other words, any approximation algorithm will always have some input graph for which it can find no coloring which uses less than twice as many colors as actually needed.

The primary reference for the material in this section is Garey and Johnson[Ref. 5]. It is highly recommended for the reader who is interested in other NP-complete problems and further discussion on this topic.

III. THE ALGORITHMS

A. GRAPH GENERATION

One of the earliest problems faced in this research was that of deciding what characteristics were desirable in the graphs to be colored and how to generate them. As the goal of the research was to contrast the efficiencies of various heuristics for optimal colorings, the graphs used needed to be, in some sense, typical, or representative of graphs which occur in real-life problems. Though the question of what is a typical graph was left largely unanswered, three possibilities for graph generation were considered.

One method considered was to use hand generated graphs with certain built-in specific features. For example, graphs could be designed with particular distributions of degree of the vertices. Another method would be to generate graphs with a given chromatic number but with varying numbers of vertices or edges. The major drawbacks to this method were the necessarily small sample of graphs and the failure of the graphs to be typical in any sense. This method was therefore not used for any of the experiments done but was used extensively for program testing and debugging.

Another possibility, quickly discarded, was to survey some of the disciplines where graph colorability models the

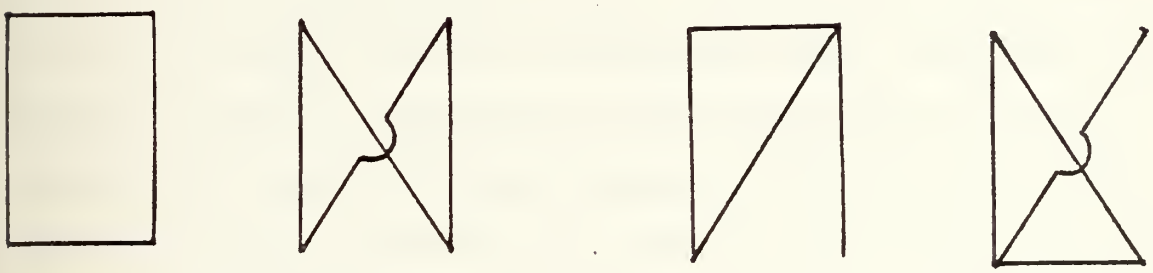
problem and to use some actual graphs which occur in real-life. Though this might generate some interesting results for those specific applications, it would also lead to results which were heavily flavored by the problem being considered. What was needed was a selection of graphs which were, in some sense, representative of the set of all graphs.

The most promising method of graph generation was to use the computer and to create a large population of random graphs on which to test the various heuristics. There is some benefit to using random graphs since one generally feels comfortable that these random graphs are fairly representative of the set of all graphs.

The primary drawback to random generation of graphs is that the generator will tend to generate more of the types of graph which have many isomorphisms than the type with few isomorphisms. A graph G is isomorphic to graph H if the vertex numbers of G can be permuted in such a way as to make the resulting adjacency matrix of G identical to the adjacency matrix of H. Fig 4 shows 2 of the isomorphic graphs for each of the only 2 distinct graphs possible for 4 vertices and 4 edges.

Since graph coloring algorithms can be influenced by the different types of graphs, a random generator may bias the

investigation of the relative performances of different coloring programs. The generation technique could be amended by somehow reducing the probability of using a particular graph by dividing the probability by the number of the graph's isomorphisms. But determining the number of isomorphic configurations of a given graph is not a straightforward problem and beyond the scope of this research.



Examples of the isomorphs of 2 distinct graphs.
Figure 4

Another technique might be to record each graph and to test each new one generated against all previous to determine whether it is an isomorphism of an already used graph. But the problem of just testing two graphs against each other is probably NP-complete(though considered an open question at this time) [Ref. 5].

In order to get on with the research goals at hand, it was decided to accept a random graph generator which generates graph type G more than type H if G has more isomorphisms than H. This is an admitted shortfall in this

research effort and it is hoped that the algorithms experimented with are not discriminated against by the shortcoming of this random graph generator.

Inasmuch as unconnected graphs are no more difficult to color than their largest connected component, it was decided to run the initial tests on only connected graphs. Two methods of generating connected graphs were considered: 1) To randomly add edges until the graph was connected and 2) to create a spanning tree of n vertices and n-1 edges(the minimum to connect the n vertices) and then add additional edges at random. Since method one would require testing for connectivity after each edge and also would fail to generate very many sparse graphs(found often in real problems) method 2 was chosen and implimented using the following algorithm:

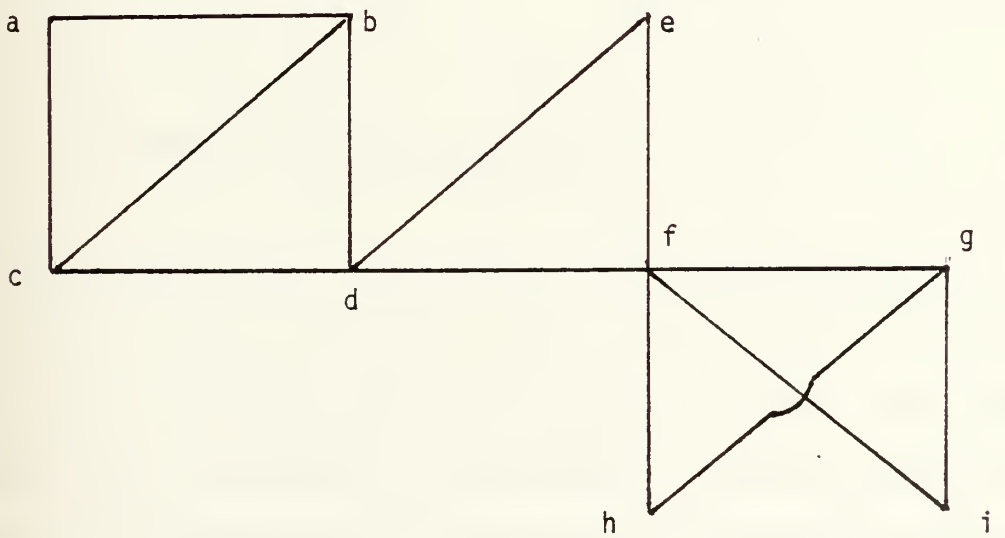
```
                    Algorithm GENGRAPH
1. Let VC be a set of connected vertices, initially empty
   Let VU be a set of unconnected vertices, initially
        VU = {1,2,...,n}
2. Choose a random vertex a
3. Move a from VU to VC
4. While VU is not empty
    choose a random vertex a from VC
    choose a random vertex b from VU
    record an edge from a to b
    move b from VU to VC
5. While Number edges < Number desired
    choose random vertex a
    choose random vertex b
    While edge (a,b) exists or if a = b
        increment a until a = n then increment b and set a = 1
        if a = n and b = n then set a = 1 and b = 1
    record edge (a,b)
End
```

At a later point in the research, it was decided that for truly valid test results, the graph generator must generate purely random graphs with no requirement for connectivity. It was discovered that the connectivity requirement changed the characteristics of the sparse graphs and non-representative graphs were being used. Therefore, a much simplified generator was used which ignored steps 1 through 4 of GENGRAPH and merely added random edges until the required number was present. Another algorithm called DIVIDE (to be described in a later subsection) was then called to divide the graph into its connected components.

The random number generator used is identical to Grogono [Ref. 9] with a modification to return integers in any range desired. The listing for both GENGRAPH and the Random number generator is enclosed in appendix A.

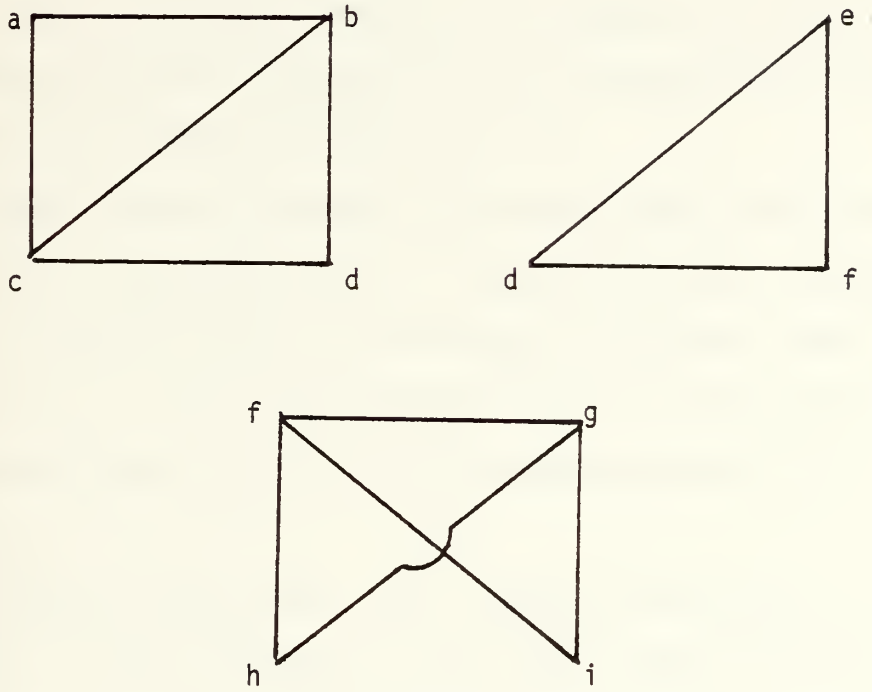
B. GRAPH DIVIDING

It has already been shown that an unconnected graph may be colored by dividing it into its connected components and separately coloring each component. A further simplifica/ tion can be made, however, if even the connected components are divided at their articulation points, if any exist. Recall that an articulation point is a vertex which is in every path from some vertex x to some other vertex y. If the articulation point is divided into 2 vertices such that the path from x to y is cut, then the graph is divided into disconnected subgraphs. For illustration purposes consider figs. 5 and 6.



A Graph with articulation points.
Figure 5

The articulation points are vertices d and f. The biconnected components are shown in fig. 6.



A Graph Divided at its articulation points.
Figure 6

In communications and transportation problems the identification of articulation points is very important because these are the 'choke' points, or places where the network becomes most vulnerable to failures. Since dividing a graph at its articulation points results in biconnected components which can be colored separately, it is also important to discover the articulation points in the colorability problem. The number of colors needed will be exactly the same

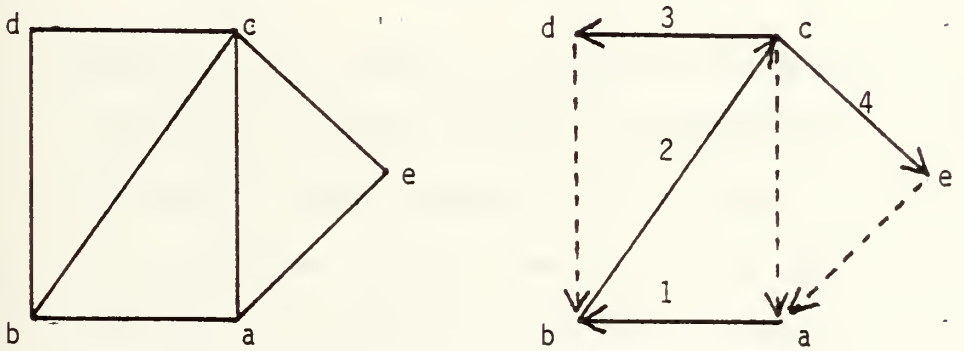
as the number needed to color the component which uses the most colors. If the actual color assignments are needed, then the graph can be 'pieced' back together one component at a time. As each component is added, the color classes are permuted so that the articulation vertex is in the same class in both components of the graph.

An algorithm to find the articulation points and the biconnected components of a graph utilizes a depth first search of the vertices in the graph and makes use of the fact that v is an articulation point if and only if there are two vertices x and y such that every path from x to y includes v.

The idea of the depth first search is to visit all the vertices of the graph by starting at some arbitrary start vertex s and recursively visiting every other vertex. Given that the search is at some vertex a, it follows an edge (a,b) to vertex b. If b has already been visited, the search returns to a and tries another edge. If b has not been visited, then the same method is applied recursively to b. At the point where all the edges incident on a have been examined, the search backs up along the edge that took it to a. The search terminates at the point where it tries to back up from the start vertex. If the graph was not connected, then some new start vertex is chosen from the un-

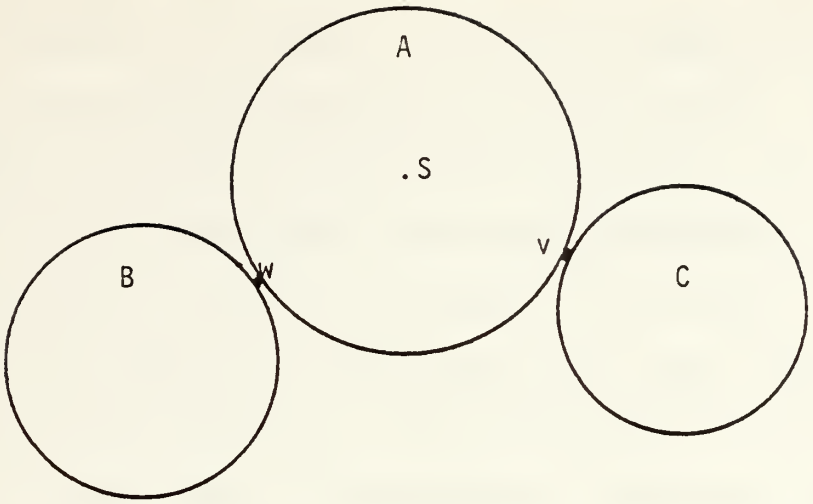
visited ones and the procedure repeated until all vertices have been visited. The edges which take the search from one vertex to a vertex which has previously been visited are called 'back edges.' Edges which go to an unvisited vertex are called 'forward edges.' These definitions will be needed for the algorithm description.

Figure 7 shows a graph with 5 vertices and 7 edges. If the depth first search is started at vertex a then the solid arrows represent a possible set of forward edges and the dashed arrows are the back edges. The labels on each edge indicate the order in which they are traversed.



A Depth first search of a Graph.
Figure 7

The basic idea of using a depth first search to find the articulation points of a graph can be seen by studying fig. 8. This schematic represents a graph with 3 biconnected components labeled A, B, and C which are joined at articulation vertices v and w. S is an arbitrary start vertex in A.



A Schematic of a graph with 3 biconnected components and 2 articulation points.
Figure 8

If the depth first search is started at vertex S in A, it will eventually enter into C through vertex v. By the depth first nature of the search, all the edges in C will be traversed before the search backs up through v. If the edges are placed on a stack as they are traversed, then when the search gets back to v, all the edges in C will be on top of the stack.

In order to recognize the articulation points, note that no vertex in C will have a back edge to any vertex which was visited prior to v. So, if the vertices are numbered as they are visited and each vertex is tagged with an index

equal to the smallest number of any vertex which can be reached through any number of forward edges and one back edge, then the articulation points can be found. When the search backs up from b to a along edge (a,b), if the index tagged to b is not less than the number assigned a, then a is an articulation point(or the root of the tree). In other words, no vertex below a in the tree has a back edge to a vertex visited before a. The following algorithm by Baase[Ref. 6] was used to find both the connected components and the biconnected components of a graph:

Algorithm DIVIDE

```
SV = stack of vertices
SE = stack of edges
top = top element of SV
Number = array to number the vertices in the order visited
Back = array to record the last vertex reachable
       through a back edge.

Algorithm:
1. initialize Number(i) = 0;
2. choose arbitrary vertex S
3. Number(S) = 1
4. Num = 2     // next number
5. stack S on SV (top = S)
6. while there exists an unprocessed edge from top to v do
    stack edge (top,v) on SE
    if number(v) > 0 then back(top) = min(back(top),back(v))
    else do
        number(b) = num
        num++
        back(v) = number(v)
        stack v on Sv
        end
    end
```

```
7. if there is more than one vertex on SV
    let v = second from top
    if back(top) >= Number(v) then
        let edge (a,b) be the top edge on SE
        while Number(a) >= Number(v) and Number(b) >=
Number(v)
            pop SE
    else Back(v) = min(Back(v),Back(top))
    pop SV
    goto 6 END
```

The code used to impliment this algorithm is enclosed in appendix A.

C. GRAPH COLORING ALGORITHM

The essential idea of the basic coloring algorithm is to build the search tree until the optimal coloring is found. Each node in the tree represents the placement of a vertex into a color class and the path from the root to the node represents a partial coloring of the graph. The cost of a node is the number of different colors used for that partial coloring. As the nodes of the tree are generated, by iteratively expanding from the start node, they are placed in a priority queue such that the first node in the queue is the node which represents the least number of colors used and is

deeper in the tree than any other node with the same number of colors. A node is expanded by generating all of its possible children and the node to be expanded next is always the first node in the queue. The stopping point for the algorithm is thus the point at which the highest priority node is a full coloring of the graph. At that point no other node in the tree could possibly be expanded in any way which results in a coloring using fewer colors. The chromatic number of the graph is then known to be the number of colors used in that coloring. Note that if all possible colorings are desired which use the optimal number of colors, then the algorithm need only save the chromatic number and to continue expanding nodes until the cost of the highest priority node is greater than the chromatic number. At each point where a node is generated which is a full coloring, it can be printed out and all optimal coloring assignments will thus be generated.

There are 2 primary data structures used in this algorithm. These are the TREE and QUEUE data structures. The elements of TREE are trees made up of nodes and the links between them which organize the nodes into a tree structure. A node N represents a color assignment to one vertex of the graph. The path from N to the root of the tree represents a partial coloring of the graph and the cost assigned to node

N is the number of colors used in that partial coloring. Each node has a pointer to its parent in the tree in order to maintain the links between nodes. There are 2 operations defined for TREE. The operation CREATE generates a tree with only one node called the root. The operation EXPAND takes as input a tree T and a particular leaf node N of T and returns a tree T which is the original tree T plus all the possible children of node N.

The nodes of the tree are implimented using an array of records where each record contains the following 5 fields: 1) the depth, or level in the tree at which it is located, 2) the cost assigned to the node, 3) the vertex number which was colored to create the node, 4) the color assigned to that vertex, and 5) a pointer to the node's parent in the tree. The operations on the TREE are implimented in the algorithm COLORALL and EXPAND as follows:

```
COLORALL            (Color all vertices of the graph)
1.  Get a node N
2.  Get number of first vertex - v .
3.  Let N be the root node
      N.vertex = v
      N.cost   = 1
      N.depth  = 1
      N.color  = 1
      N.parent pointer = NULL
4. While N.depth < n do
      expand N  (see algorithm EXPAND below)
      N = first node in priority queue
5. Print out statistics,coloring,and chromatic number.
```

The algorithm to expand a node is as follows:

```
EXPAND(N)  (N is the node to be expanded)
1.  Choose a vertex v to be added to the partial coloring.
    (described in subsection D.)
2.  let C = {1,2,...,N.cost} be the set of colors used
                in the path to node N.
3.  Trace the parent pointers from N to the root.  For
    each node M in the path to the root, if an edge exists
    between vertex v and M.vertex then remove color M.color
        from C.
4.  For each color c_i left in C create a new node N_i and
    put it in the priority queue.
    N_i.vertex = v
    N_i.cost   = N.cost
    N_i.depth = N.depth + 1
    N_i.color = c_i
    N_i.parptr = N
5.  Add one new color and create one more node N_j.
    N_j.vertex = v
    N_j.cost = N.cost +1
    N_j.depth = N.depth +1
    N_j.color = N.cost + 1
    N_j.parptr = N
END
```

In terms of graph coloring, step 4 creates a child node for each of the ways in which the new vertex can be added to one of the existing color classes. Step 5 generates a node which represents putting the new vertex in a new color class by itself.

The QUEUE data structure is used to organize the leaf nodes of a tree according to their priority. There are 2 components which determine the priority of a node. The first is, of course, its cost as defined earlier. The second component is used to determine the highest priority

between 2 nodes with the same cost. When more than one node is available which have the same cost then the node which is deepest in the tree has a higher priority since it is closer to a complete coloring. There are two operations defined for the QUEUE called ADDNODE and REMOVETOP. ADDNODE takes a queue Q and a node N and returns a queue Q with N in its proper priority location. REMOVETOP takes a queue Q and returns the node N with highest priority.

The priority queue is implimented as a heap, as described in ref. 10. The nodes are organized into a full binary tree such that the root is the node with highest priority and every node in the tree has a higher priority than either of its children. When adding a node to the tree, it is put in the first available location and then 'sifted up' until its parent has a higher priority. When the root node is removed, the last node is moved to the root position, then 'sifted down' until both its children have a lower priority. The binary tree is maintained by an array of pointers q() where q(1) points to the node at the root of the binary tree and the children of q(i) are located at q(2i) and q(2i + 1).

In any tree search algorithm it is important to ensure that isomorphic paths are not generated. For the problem of graph colorability, two colorings are isomorphic if the ver-

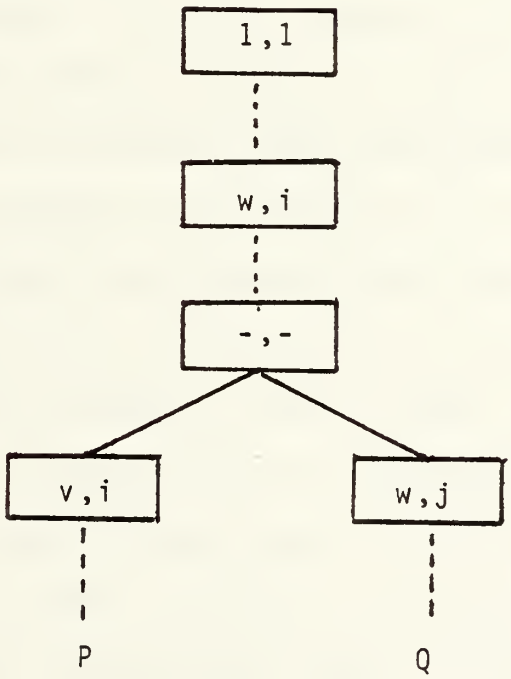
tices of the graph are partitioned into color classes in the same way in both colorings. Therefore, if one coloring uses k color classes, then there are k! possible isomorphs of that coloring. In order to prevent the tree from containing all of these unneeded paths, one must either check for possible isomorphs with the addition of each new node or prevent isomorphic path generation from occuring by the design of the algorithm.

There are two essential ingredients of algorithms COLORALL and EXPAND which ensure no isomorphs: 1) When a node N is expanded, at most 1 new color is added to the available color set, and 2) In expanding a node N, only 1 vertex of the uncolored vertices can be added to the partial coloring and that vertex must be the same for all the children of N.

THEOREM: Any tree search graph coloring algorithm with the above 2 properties will not generate isomorphic colorings.

Proof: Assume the algorithm generates a tree with 2 isomorphic paths P and Q. Then, by the definition of isomorphic colorings, for each color class in P there exists a color class in Q with exactly the same elements. Let vertex v be the first vertex colored in P and Q such that the color of v in P is different from the color of v in Q. Call the class

containing v in path P, C_i^p and the class containing v in Q, C_j^q . If the subscript indicates the color, then $i \neq j$. Since only one new color could be added when v was colored, then at least one of C_i^p or C_j^q contains a vertex w which was colored prior to v. Without loss of generality, let that class be C_i^p. In other words, v and w are in the same color class in path P. But, since v was colored differently in path Q and w is the same in both P and Q, there can be no color class in Q containing both v and w. Therefore, P and Q are not isomorphic colorings. Figure 9 illustrates this proof.



Two Paths P and Q which are not isomorphic
Figure 9

D. THE VERTEX ORDERING FUNCTIONS

The purpose of this research was to investigate different branching functions in a branch and bound approach to solving the chromatic number problem. For the particular branch and bound algorithm used, the branching function is the order in which the vertices are selected to be added to the partial coloring at each new level of the tree. In this research, five different methods of ordering the vertices were investigated. Four of the methods were static orderings in which the vertices were preordered and then the graph coloring algorithm was called to search the entire tree. In every path through the tree in these methods the order of choosing the next vertex was constant. In the fifth method, the vertex chosen to be colored next was dependent on the particular color assignments made prior to the node to be expanded. This method of dynamic ordering results in a larger time expenditure at each node since the order must be recomputed at each expansion of the tree.

Note that the purpose of any of these orderings is to make the search tree as small as possible, thus requiring fewer nodes and fewer expansions. If the amount of work done to expand a node (create its children) is constant for each method, then the resulting size of the search tree is a direct measure of the benefit of any procedure. Since the

height of the tree will always be the number of vertices in the graph, the only way to decrease the number of nodes is to decrease the branching factor (number of children of each node). The optimal branching factor is, of course, 1. With the algorithm COLORALL, this is reached when each vertex to be colored requires a new color. It is approached most closely when the vertices are ordered such that the next vertex is always the one with the fewest colors available to be colored. Three of the static orderings attempt to accomplish this ordering through the use of some heuristic function to order the vertices. The dynamic ordering actually determines, at each node, which of the uncolored vertices has the fewest colors available.

Obviously, any graph which is complete(a clique), will be colored with a branching factor of one, no matter what order is used. The same applies to a graph which is just a forest of vertices(no edges). The difficulty lies in the midrange. Another way to look at it is that coloring is easiest when there are few restrictions(edges) or very many restrictions to the colors which can be used. Each of the ordering methods, therefore, really attempts to find the vertices with the most restrictions and color them first. Thus, the tree has the smallest branching factor at the top

and more of the tree is pruned off resulting in fewer nodes to search.

Following is a description of each ordering. The listing for all orderings is in the program ORDERNODE in appendix A. The case numbers in the switch statement correspond to the numbers shown here.

1. Random

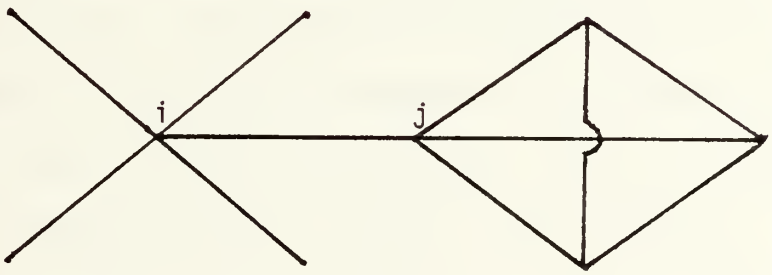
This method of selecting vertices was used as a baseline against which to compare any benefit of the other methods. Inasmuch as the graph was generated randomly, no real work needed to be done. The vertices were merely chosen in the order of increasing vertex number.

2. Degree

This method, used in the Welsh-Powell approximation algorithm[Ref. 11], orders the vertices by decreasing values of the degree of the vertices. Recall that the degree of a vertex is the number of edges incident upon it and is easily calculated by summing the elements of the rows of the adjacency matrix. The idea behind this method is that the vertex with the highest degree has more restrictions on it when choosing a color so it should be colored first. That color used is then removed from the colors available for every vertex connected to it.

3. Dvec 3

DVEC 3 is the name given by this author to a method devised by M.R.Williams[Ref. 4]. In the prior method of ordering the vertices by degree, the underlying premise is that if a vertex i is connected to p other vertices, then it will be more difficult to color than vertex j which is only connected to q vertices ($p < q$). What that method fails to consider is the degree, or difficulty, of the vertices to which i and j are connected. For example, if vertex j is in a clique of size 4 and i is the center of a star with 5 points, as shown in fig. 10, then certainly j should be colored before i, as well as all the other vertices in the clique.



Example of Shortcoming of Degree method
Figure 10

Williams recognized that the vector of degrees of the vertices was the first step towards the development of the dominant eigenvector of the adjacency matrix. Recall that

$$d_i = \sum_j a_{ij}$$

In the standard technique, d_i^{∞} is the eigenvector corresponding to the largest eigenvalue of matrix A and is found by iteration using:

$$d_i^{n+1} = \sum_j a_{ij} * d_i^n$$

If d_i^0 is the vector of degrees, then this scheme does take into account the degrees of the vertices to which a vertex is connected. And, in fact, the number of iterations used determines how far away from vertex i to look in calculating the relative difficulty of coloring vertex i.

Williams also determined that the relative order of after a few iterations. This fact was borne out by results of this research. Although improvement was seen out to 2 or 3 iterations, little or no changes in the size of the search tree was found after 3 iterations. The only effect was the increased computation to calculate the vectors. For this reason, 3 iterations was chosen as a good number to use to compare this method against the others.

4. Adjacency

The rationale behind this ordering method is that the vertex which is adjacent to the most previously colored vertices will probably have the fewest colors available.

This is not necessarily true, however, as a vertex may be adjacent to many vertices which are all in the same color class, thus reducing its available color space by only one color.

The algorithm to determine which vertex is next, considers the rows in the adjacency matrix for all the previously colored vertices, and adds the columns. The highest column total then indicates which vertex is to be colored next. Previously colored vertices can be eliminated by subtracting an arbitrarily high number from the running column total, for each vertex, when it is chosen. In the case of ties, the vertex of highest degree is chosen.

5. Least colors available (LCA)

In all the previous static orderings, an assumption was made that the best vertex to color next, at any point in the tree, was independent of the specific color asignments made up to that point. We were simply trying to find the vertex that was 'most probably' a better choice than the others. In this dynamic ordering, when expanding each node, we actually choose the vertex which will result in the fewest children being created, thus minimizing the branching factor. Since a child node is created for each available color which the next vertex can be colored, we must consider all uncolored vertices in the path to the node being expand-

ed and determine which vertex has the fewest colors available. To do this note that ideally we would like to choose a vertex which is adjacent to at least one vertex in each color class used so far. Thus a new class would have to be created and there would be only one child node created. Short of this ideal situation, we would like to choose the vertex which is adjacent to at least one vertex in most of the color classes so far created.

This is accomplished by creating a new matrix CC similar to the adjacency matrix such that there is a row in CC for each color class generated and a column for each vertex. Then we define

$$cc_{ij} = \begin{cases} 1 & \text{if } (v,j) \in E \text{ for some vertex } v \text{ in color class } i. \\ 0 & \text{otherwise.} \end{cases}$$

i.e., $cc_{ij} = 1$ iff vertex j cannot be colored color i.

The matrix CC is easily created from the adjacency matrix A as follows: Let cc_i be row i in CC, and a_i be row i in A, both represented as bit strings.

```
initialize all cc_ij = 0
for i from 1 to NUM-COLOR-CLASSES
    for each vertex j in color class i
        cc_i = cc_i V a_i
```

The next step is to generate a vector called SUM such that

$$SUM_j = \begin{cases} \sum_i cc_{ij} & \text{if } j \text{ is uncolored} \\ -1 & \text{if } j \text{ is colored} \end{cases}$$

Then the value of j for which SUM is a maximum is the vertex which has the fewest color classes available to which it can be added. Again, for consistency, ties are broken by choosing the vertex of highest degree.

E. CONTROLLING ALGORITHM

Inasmuch as the purpose of this research was to experiment with various orderings of the vertices, the flow of control was fairly straightforward. After only a few trial cases it was discovered that the primary limitation to the size of the graph which could be colored was computer storage space, not time. As the graphs got larger, more nodes were required in the search tree until the available memory was used up at about 4000 nodes. Although unsigned variables were used in the node structures and space was conserved wherever possible, this limitation could not be significantly changed. For even the best orderings, this meant graphs of about 30 vertices or less. For empirical data then, graphs were generated with from 10 to 30 vertices and edges from 20% to 80% of the maximum possible. For each graph size, 100 graphs were generated (some of which were probably identical and many which were at least isomorphic) and each ordering method was used to color the graph. The graph was then divided at its articulation points and each biconnected subgraph was colored using each ordering func-

tion. Data about the size and shape of the search tree was collected and tabulated for all the above samples (appendix B).

Note that 100 trials of each graph size is a relatively small sample, but was a necessary restriction to keep the entire program execution time to within a few hours.

The following algorithm shows the setup used for obtaining the experimental data.

algorithm MAIN

```
initialize
For n from minsize to maxsize    (n is number of vertices)
begin
   compute maxedges = (n)(n-1)/2
   For fullness from 30% to 70% of maxedges
   begin
      compute number of edges e = fullness * maxedges
      For trials from 1 to 100
      begin
         generate random graph (n vertices, e edges)
         For each ordering method
         begin
            color the graph
            record data (expansions, nodes, branching factor)
            divide the graph
            For each biconnected component of the graph
            begin
               color the component
               record component data
            end
            sum the component data
         end
      end
      compute averages of all data collected
      print out data
   end
END
```

The function MAIN in appendix A is the controlling algorithm for the entire process.

F. GENERAL COMMENTS ON ALGORITHM DEVELOPMENT AND THE PROGRAMMING LANGUAGE 'C'

In all the programming done during this research, efforts were made to save time and storage space whenever feasible. The first attempt to design a coloring algorithm used a backtracking method which is common for tree searches. However, the size of the search tree was so large that the storage required to stack the variables used in each recursive call to the backtracking function was exorbitant. For that reason, an iterative scheme was chosen and implimented.

Further savings were made by using bit strings and 1 bit variables whenever possible. Since the maximum graph size was to be less than 32 vertices, the long integer variables of 32 bits in 'C' were ideally suited for the adjacency matrix and many other needed variables. Also 'C' has a good assortment of bit manipulation operators including multiple shifts, or, and, and exclusive or. Using these led to time savings in many cases.

Space was conserved by using the unsigned variable type in arrays and records. With this convention, the length of the storage cells reserved were only as long as needed. For

example, the vertex numbers from 0 to 31 could be stored in only 5 bits. Another practice, not necessarily considered good programming technique was the fairly extensive use of global variables. Though risky, it did save the time required to pass parameters and the space to store local variables.

Overall, this author feels that 'C' is a very appropriate language for research of this type.

IV. RESULTS

A. THE PARAMETERS

In attempting to determine the relative 'goodness' of several computer programs, one must first find some measurable parameters to be minimized or maximized to indicate the best program. For approximation algorithms, one is very often not interested in how long a program takes as long as it is some linear function of the problem size or possibly even O(nlgn). What one measures is the closeness of the approximation to the actual optimal solution.

For this research, the solution to the graph coloring problem was to be the exact chromatic number in every case. The coloring algorithm was to find the chromatic number of a given graph and the 'goodness' of the vertex ordering methods was to be determined by how fast the algorithm ran and how much space was used. The specific questions to be answered were: 1) Is there a best-time ordering for all graphs?, 2) Is there a best-space ordering for all graphs?, 3) Are they the same?, 4) What type, or size, of graphs favor which ordering if there is a difference?, and 5) Is one ordering better on the average, but another better for worst-case situations.

The results of the many experiments which relate to these questions is tabulated in appendix B. The discussion

here is of a general nature and will summarize the results. There were basically two separate categories of measurements made - those related to time and those related to space. Though they are closely related in this problem, the parameters measured are distinct and will be discussed separately.

1. Time

In determining the difference in time-complexity of the 5 coloring methods, there are 2 modules of the program to consider - ORDERNODE and COLORALL(of which EXPAND is a part). These functions are the only two whose time-complexity is a function of the vertex ordering as well as the graph size.

For the 4 static orderings the time-complexity is easy to determine. For consistency, each ordering merely filled an array called next, where next(i) was the vertex to follow i in the ordering. Random is therefore clearly O(n) since the vertices are ordered by vertex number. One traversal through the array next is all that is needed to fill it. DEGREE is $O(n^2)$ since the adjacency matrix has to be traversed to find the degree of each vertex, then a simple sorting of the vertices is O(nlgn) so the overall complexity is $O(n^2)$. DVEC 3 is $O(n^2)$ if the number of iterations is kept constant(such as 3 in this case). It would be conceivable to iterate n times for the best possible order-

ing in which case DVEC n would be $0(n^3)$. ADJACENCY is also $0(n^2)$ because a running total was kept of the relative position of the unselected vertices as each new one was determined. Thus the adjacency matrix was addressed in only one row for each new vertex. For the dynamic ordering LCA, no work was done in the function ORDERNODE but rather in the function PICKNODE which was called every time a node was to be expanded. PICKNODE itself has a time complexity of n^3 since it takes n^2 operations to create the n x n matrix of color classes CC and each element in CC requires 0(n) to compute. But the bigger concern is that the time-complexity of the branching function EXPAND is n^3 times as great for LCA than for the static orderings.

For the static orderings the time-complexity of generating the search tree has 2 components - the time to expand a node and the number of expansions needed to find a solution. For LCA, a third component, the time to find the next vertex, must also be considered.

Once the next vertex is found, the time required by the function EXPAND is 0(n). The only iterative part of that function is following the path to the root and then generating a child node for each possible coloring. Since the depth of the tree can be up to n-1 and the number of children can be n, this function is 0(n).

So far, all the time complexities have been polynomial. If this were also true of the number of expansions, then one could claim that the overall algorithm would run in polynomial-time since the product of any number of polynomials is also a polynomial. Unfortunately, that is not the case for worst-case situations. Consider for a moment the size of the full search tree generated by this algorithm if no pruning is allowed. Refer to figure 11 which shows the first four levels of the search tree. The notation x,y in each node of the tree indicates that the node represents an assignment of color y to vertex x. Note that at the root there is 1 node followed by 2 nodes at level 2. Then at level three there are 5 nodes, 2 children of one in level 2 and 3 for the other. To determine the number of nodes at level k, recall that one color class is added for each level in the tree. Also, a node at level k represents one of the ways of partitioning the vertices into k or fewer color classes. Therefore, the number of nodes at level k is the number of ways to partition a set of n elements into k or fewer classes.

The Stirling numbers S(n,m) represent the number of ways to partition n elements into m distinct

THE FIRST FOUR LEVELS OF THE UNPRUNED SEARCH TREE

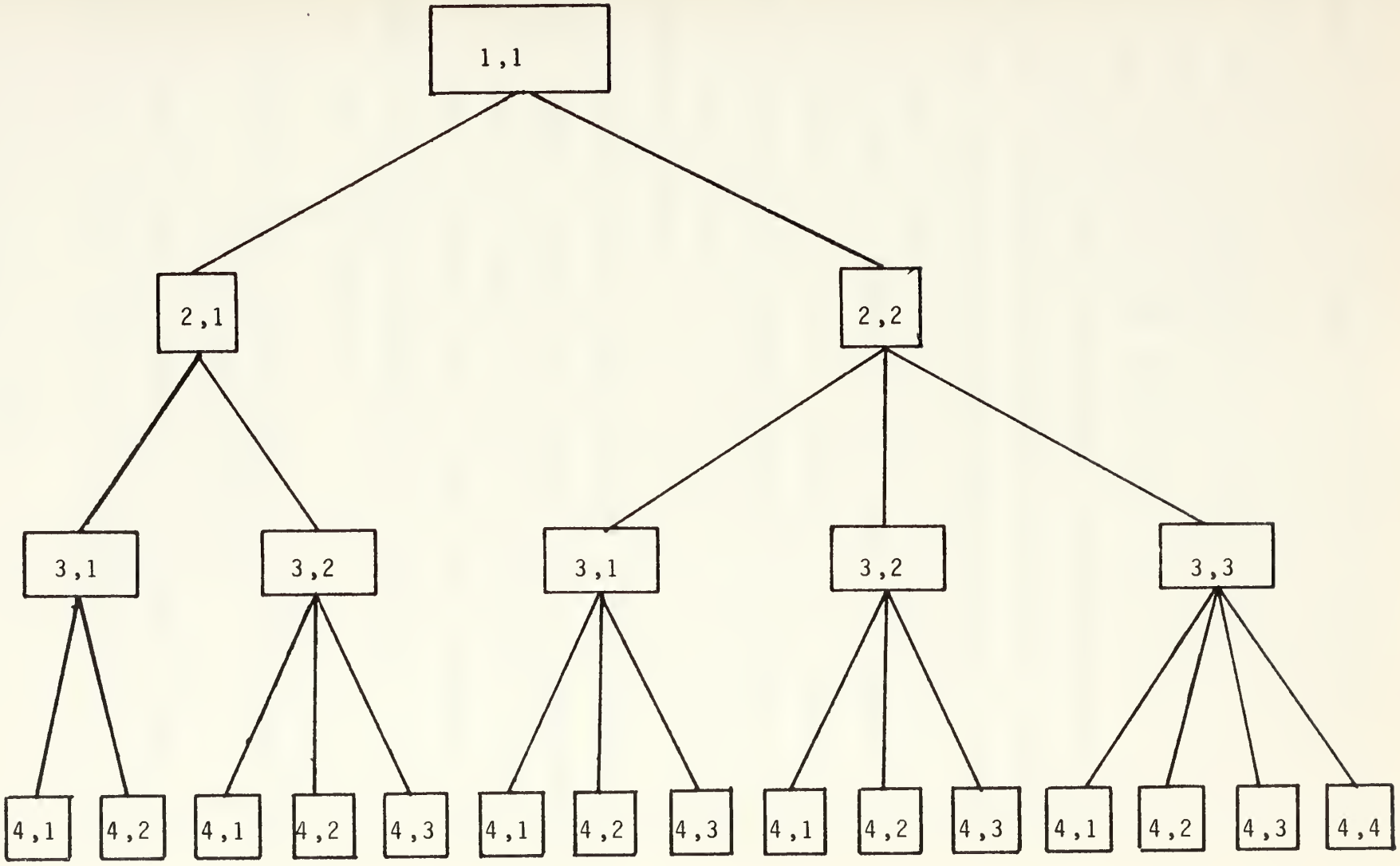


Figure 11

classes[Ref. 12] and are generated by the following recurrance relation:

$$S(n+1,k) = S(n,k-1) + k * S(n,k)$$

$$\left.\begin{array}{l} \text{where } S(n,n) = 1 \\ \text{and } \quad S(n,1) = 1 \end{array}\right\} \text{ for all } n$$

To understand this relation, consider the following: Given that n elements can be partitioned into k classes in S(n,k) ways, there are two ways to add the (n+1)st element. First, it can be added to any one of the k classes for each way of partitioning n elements. The number of ways of doing this is given by k * S(n,k). Secondly, the n elements can be partitioned into k-1 classes and then the (n+1)st element can be a single element in a new class. This is the reason for the S(n,k-1) term.

But the number of nodes at level k is the sum of the Stirling numbers S(n,m) for m from 1 to k. These numbers are called the Bell numbers B after E.T.Bell[Ref. 13]. B_n is the number of partitions of a set with n elements into any number of distinct classes up to n. That is

$$B_n = \sum_k S(n,k)$$

To see how fast these numbers grow, the first 8 are:

1,2,5,15,52,203,877,4140

In other words, at level 8 in the search tree there would be 4140 nodes and the total number of nodes in the

tree would be the sum of B_i for i from 1 to 8. To see what the upper and lower bounds on B_n are, consider again the tree in fig. 11. The fewest number of children of each node is found in the leftmost path down the tree. In this path, every node has two children. In the rightmost path down the tree, a node at level i has i children and this is the largest number of children for any node at that level. Thus the total number of nodes at any level i lies somewhere between the two cases, or

$$2^n <= B_n <= n!$$

Since B_n is growing faster than 2^n it is easy to see that as n gets large, the possible size of the tree is going to far outweigh any of the polynomials so far discussed. Obviously, unless a significant amount of pruning is done, the time to expand every node at each increasing level in the tree is going to grow far too rapidly. Since the search strategy was fixed for all orderings, the most significant way to compare the time savings of each ordering was to count the number of nodes which needed to be expanded. Therefore, the number of calls to the function EXPAND was used as the primary measure of the time complexity of the various orderings.

2. Space

There are only two data structures in these programs whose size is dependent upon the vertex ordering: the array of nodes in the search tree and the array of pointers used for the priority queue. But the number of nodes in the queue is always less than the total number of nodes in the tree since expanded nodes are removed from the queue. Since the number of expansions has been recorded for a time analysis, only the total number of nodes in the search tree at the time an optimal solution is found need be recorded for a space analysis.

In order to see the shape of the search tree, another parameter, the average branching factor, was recorded for each level in the tree. The average branching factor is defined to be the average number of children of each node in the search tree. A full binary tree would thus have an average branching factor of 2.0. If the branching factor is recorded for each level in the search tree, one can see the 'shape' or the 'narrowness' of the tree. The most desirable situation is the one in which the branching factor stays at, or as close as possible, to 1.0 in the first several levels of the tree. This indicates that the maximum pruning took place as early in the search as possible, thus reducing the search space by more nodes for each prune. The 'goodness'

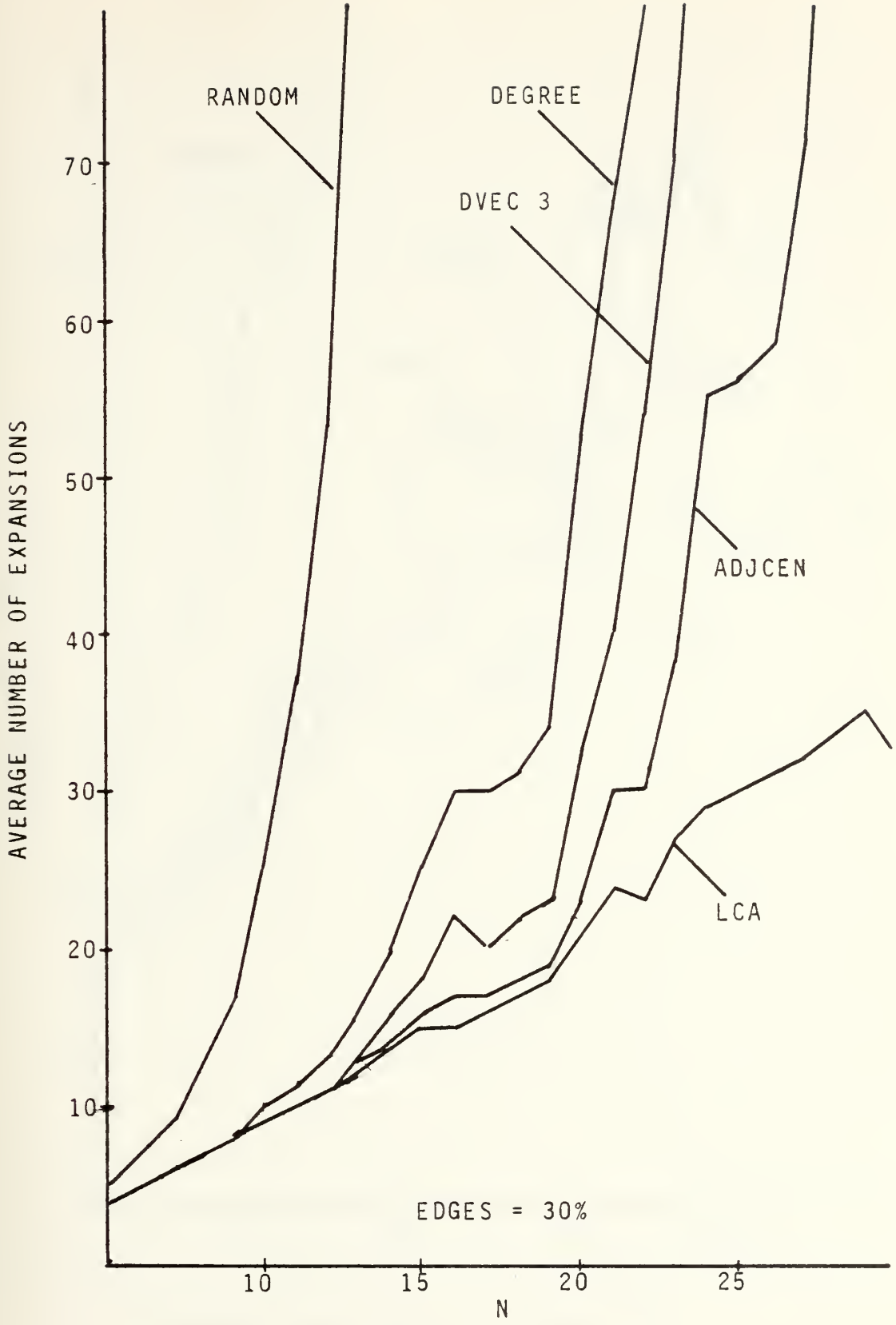
of the ordering functions can thus be seen by how long the branching factor stays at 1.0 and how low it stays when it does start to increase.

B. TIME ANALYSIS

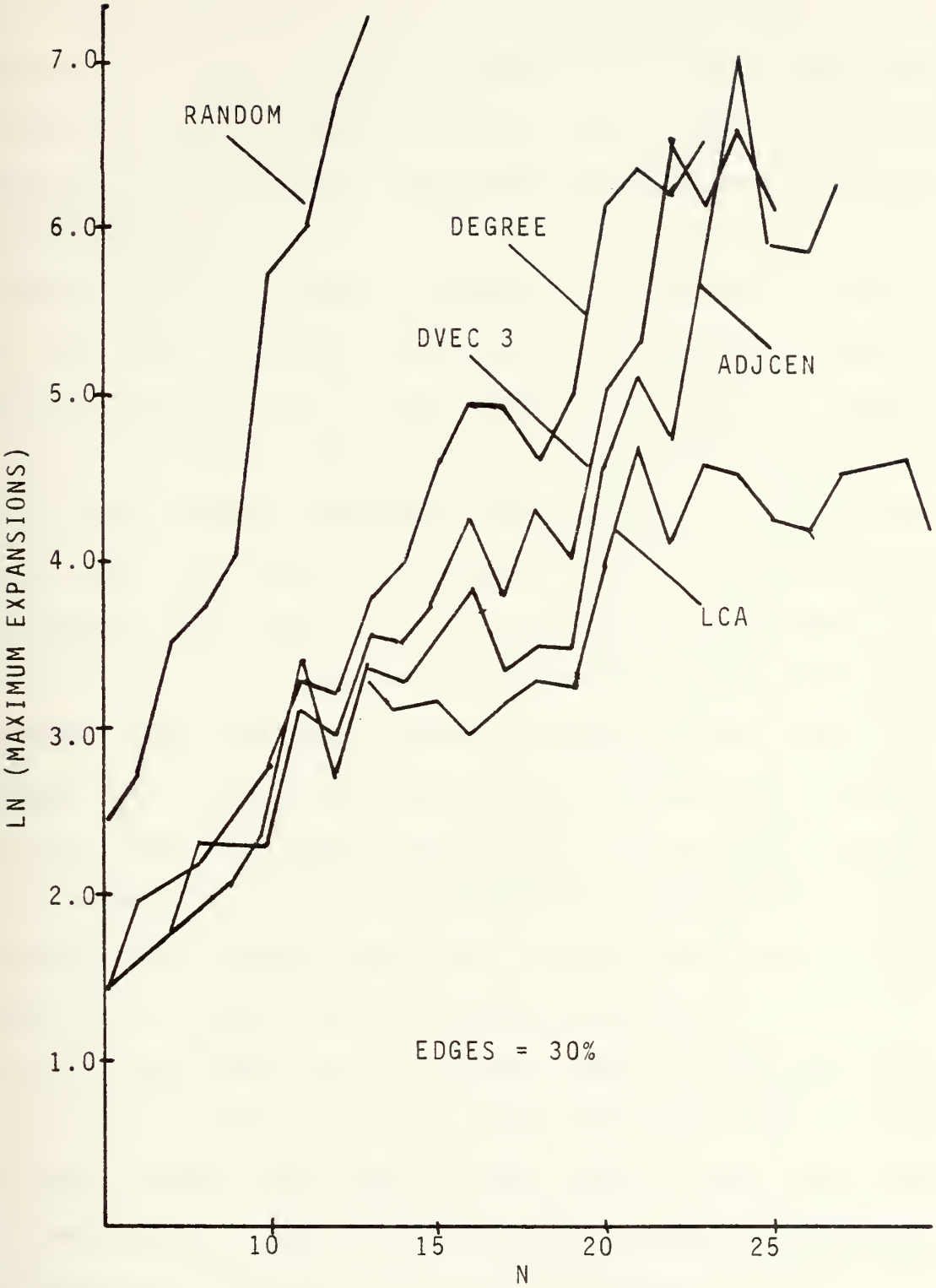
In analysing the time required by the program for each ordering method there were two cases of interest: average time, and the time required in the worst case. Figure 12 is a plot of the average number of expansions required for each ordering method when graphs containing 30% of the maximum edges were colored.

For the worst case graphs the natural logarithm of the number of expansions was plotted since exponential growth was expected. Figure 13 is a plot of the worst case graphs for 30% edges. These plots are not smooth and in fact are quite ragged because each point actually represents the coloring of one particular graph in the sample of 100. Note that the ordering methods parallel each other in the dips and hills of the plot, indicating that the same graph was usually the most difficult to color for all of the methods.

The significance of these two plots is in the total dominance of the LEAST COLOR AVAILABLE method of ordering the vertices. Though all the other methods eventually 'exploded' even in the average case, LCA stayed very close to linear for the full range of graphs up to 32 vertices.



AVERAGE EXPANSIONS VS. GRAPH SIZE
Figure 12



MAXIMUM EXPANSIONS VS. GRAPH SIZE
Figure 13

Furthermore, the worst case analysis shows the number of expansions for LCA leveling off between 20 and 30 vertices. It would be an interesting experiment to carry the graph sizes out past 32 to see if and when LCA also starts to show explosive growth.

Another result, though not shown in graphical form is that the time required by any of the ordering methods is lowest for graphs with very few edges as well as for graphs which are close to being complete. For any given value of n, empirical evidence indicated that the most time is required when the number of edges is in the vicinity of 50% (see appendix B). This is an intuitively obvious result if one considers what happens in the search tree. With very few edges, very few colors will be needed so the number of children of a node will be minimal. The search, in this case, will tend to proceed down the left side of the search tree. Most of the right side will be pruned away because the cost of the nodes in the right side of the tree will be greater than the cost of the optimal solution.

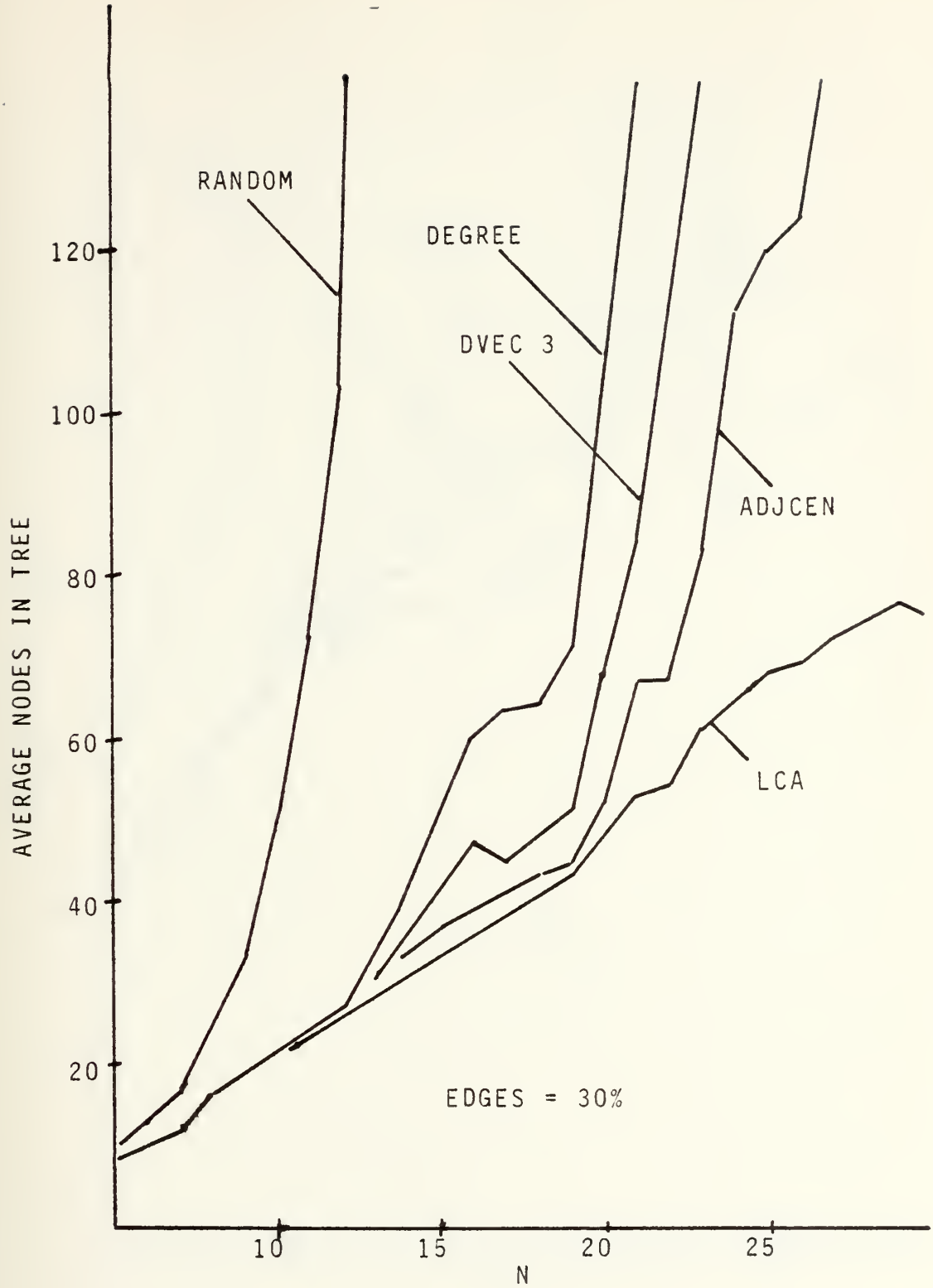
In the case where there are many edges, a new color will be required at each level and the search will tend to stay very close to the right side of the tree. The left side will be pruned away since many of the nodes on that side will represent illegal colorings due to edges.

C. SPACE ANALYSIS

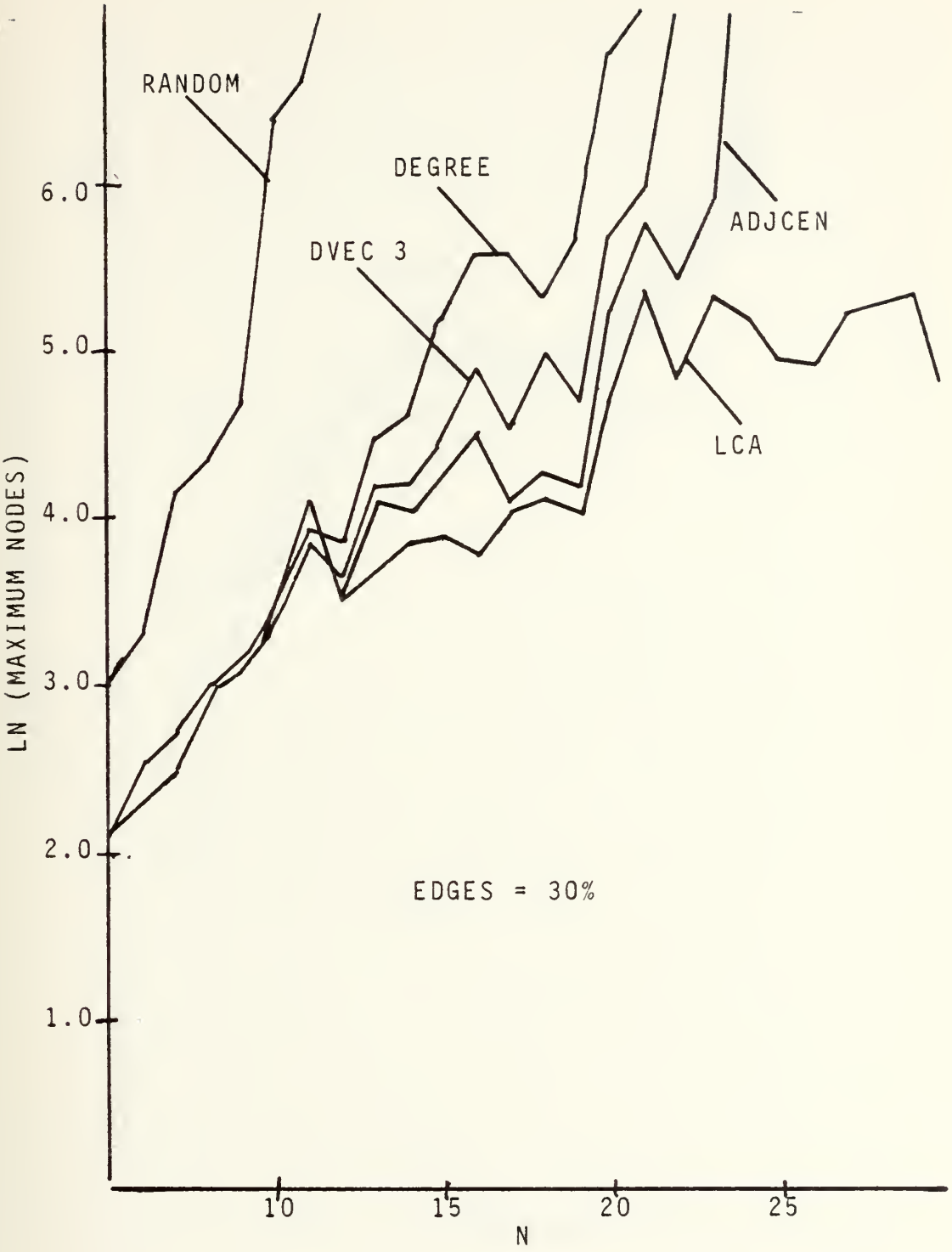
Just as for time, the space analysis is of both average and worst case situations. Figure 14 is a plot of the average number of nodes in the search tree when the first optimal solution was found. Figure 15 is a plot of the natural log of the maximum nodes needed for any of the 100 graphs. Again, the number of edges is 30% of the maximum possible.

Not surprisingly, the space analysis results are very similar to the results found in the time analysis. In every case LCA is superior in the sense that it uses less space and its growth rate is much improved over the other methods.

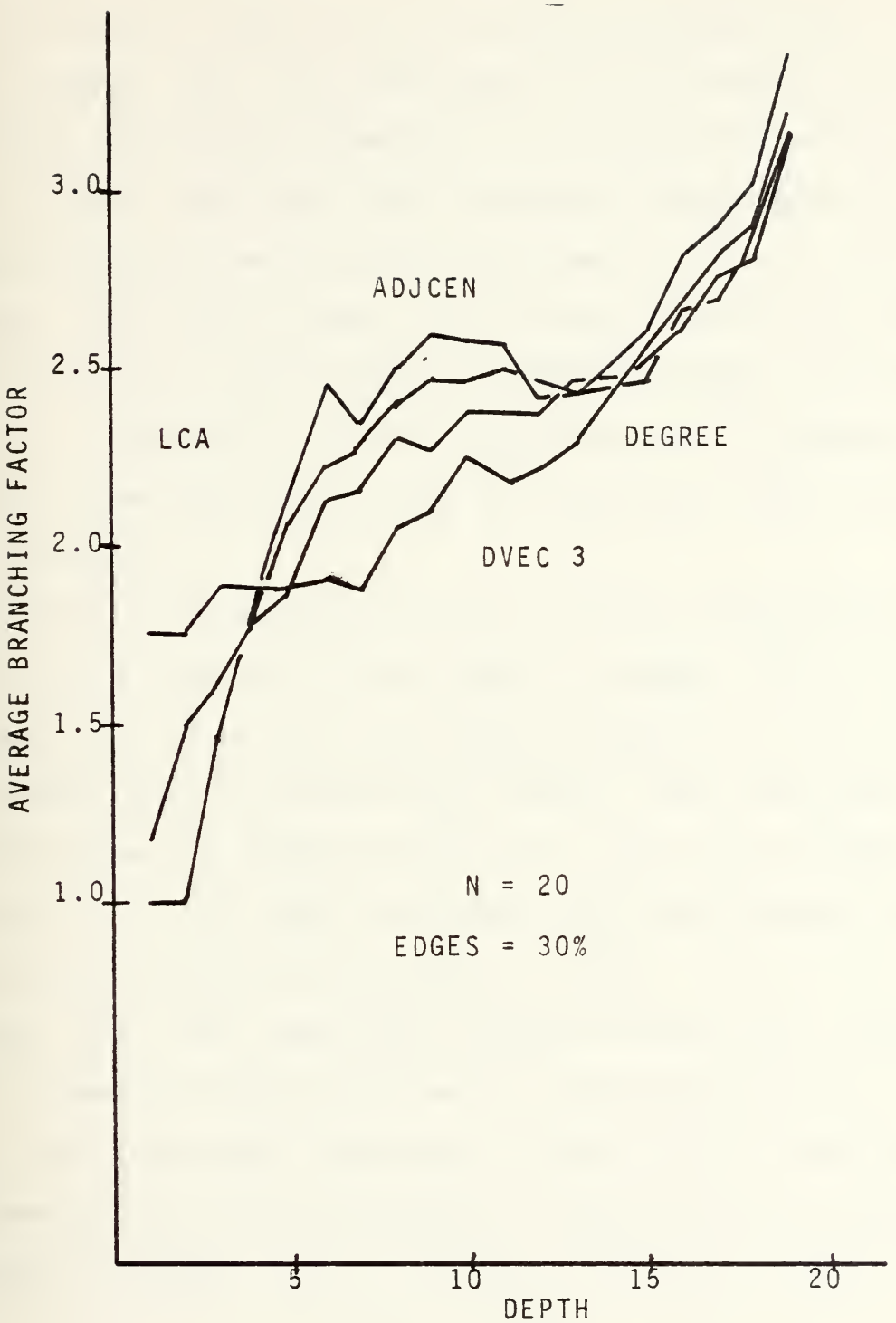
Since the relationship between the number of expansions and the total number of nodes is the branching factor, it comes as no surprise that LCA exhibits the lowest branching factor at the top of the search tree. Figure 16 shows the average branching factor as a function of depth for each of the 4 heuristic orderings. Note that LCA and ADJACENCY have the same branching factor for 3-4 levels but then ADJACENCY grows faster in the midrange. Note also that both of these methods have a higher branching factor in the middle of the tree than either DEGREE or DVEC 3. This fact shows how powerful 3 or 4 prunings at the top of the tree can be.



AVERAGE NODES VS. GRAPH SIZE
Figure 14



MAXIMUM NODES VS. GRAPH SIZE
Figure 15

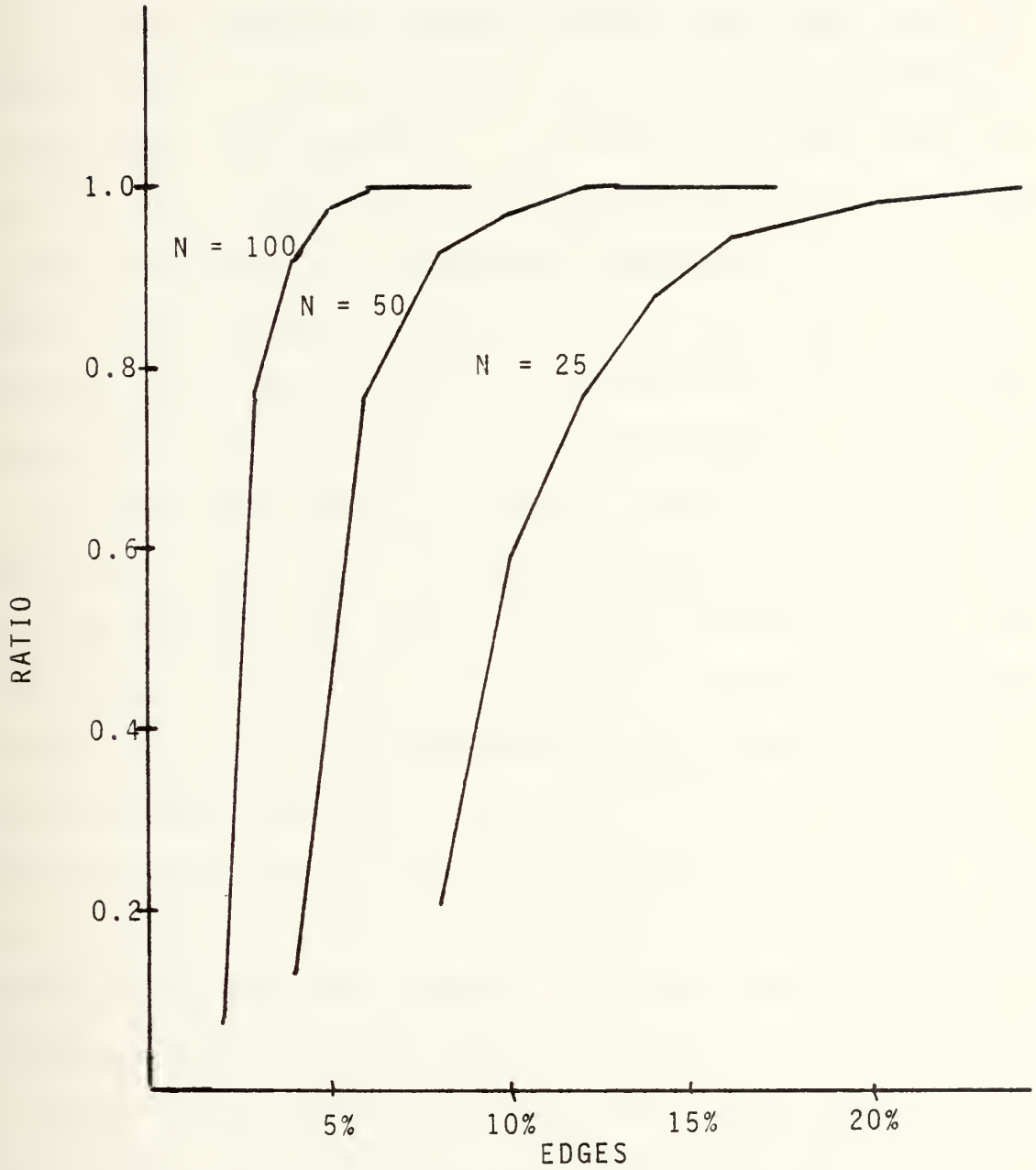


AVERAGE BRANCHING FACTOR VS. DEPTH IN TREE
Figure 16

D. ADVANTAGE OF DIVIDING THE GRAPHS

In the first stage of this coloring process, the graphs were colored just as generated with no attempt to divide them into biconnected components. In the second stage, they were divided and then each component was colored and the number of expansions required was summed up for all the components. For sparse graphs where there were many very small components, the results of this experiment were quite substantial. However, as the graphs became more dense, the original graph was biconnected and no savings were derived from dividing.

Figure 17 shows the size of the largest biconnected component as a function of the number of edges. In this plot, the ratio of the size of the largest component to the original graph size is plotted for 4 cases. Note that since the maximum number of edges grows as n^2, when the number of vertices in the graph increases, the graph becomes biconnected with a smaller percentage of edges. Thus it is reasonable that the empirical evidence gained here indicates that a graph with 100 vertices is biconnected 99% of the time with 6% edges whereas a graph with 50 vertices is biconnected 99% of the time when it has 12% of the maximum vertices included.



RATIO OF LARGEST BICONNECTED COMPONENT TO ORIGINAL SIZE
Figure 17

In investigating the effects of dividing on graph coloring, note from fig. 17 that with 30% edges, the largest biconnected component is very close to the original graph for all n. Therefore, in order for dividing to be beneficial, random graphs must be very sparse. For the empirical results shown in fig. 18 graphs were generated with from 10% to 30% edges. The benefit of dividing is then shown by plotting the ratio of the number of expansions needed for the divided graph to the number of expansions for the undivided graph against the percentage of edges present. Inasmuch as the relative benefit was the same for all ordering methods, only LCA was used for this graph.

It appears as though the benefits from dividing are insignificant since only very sparse random graphs have articulation points. But many real-life problems which are modeled by graphs may be quite dense yet have articulation points. This is true particularly in transportation and communications problems where some of the vertices may represent junctions or hubs through which all traffic must flow. Consider the city of Chicago in a nationwide transportation network for example. In these cases graph dividing may result in a much faster coloring.

Also there are problems which have graph models which are indeed very sparse. Consider, for example, the exam

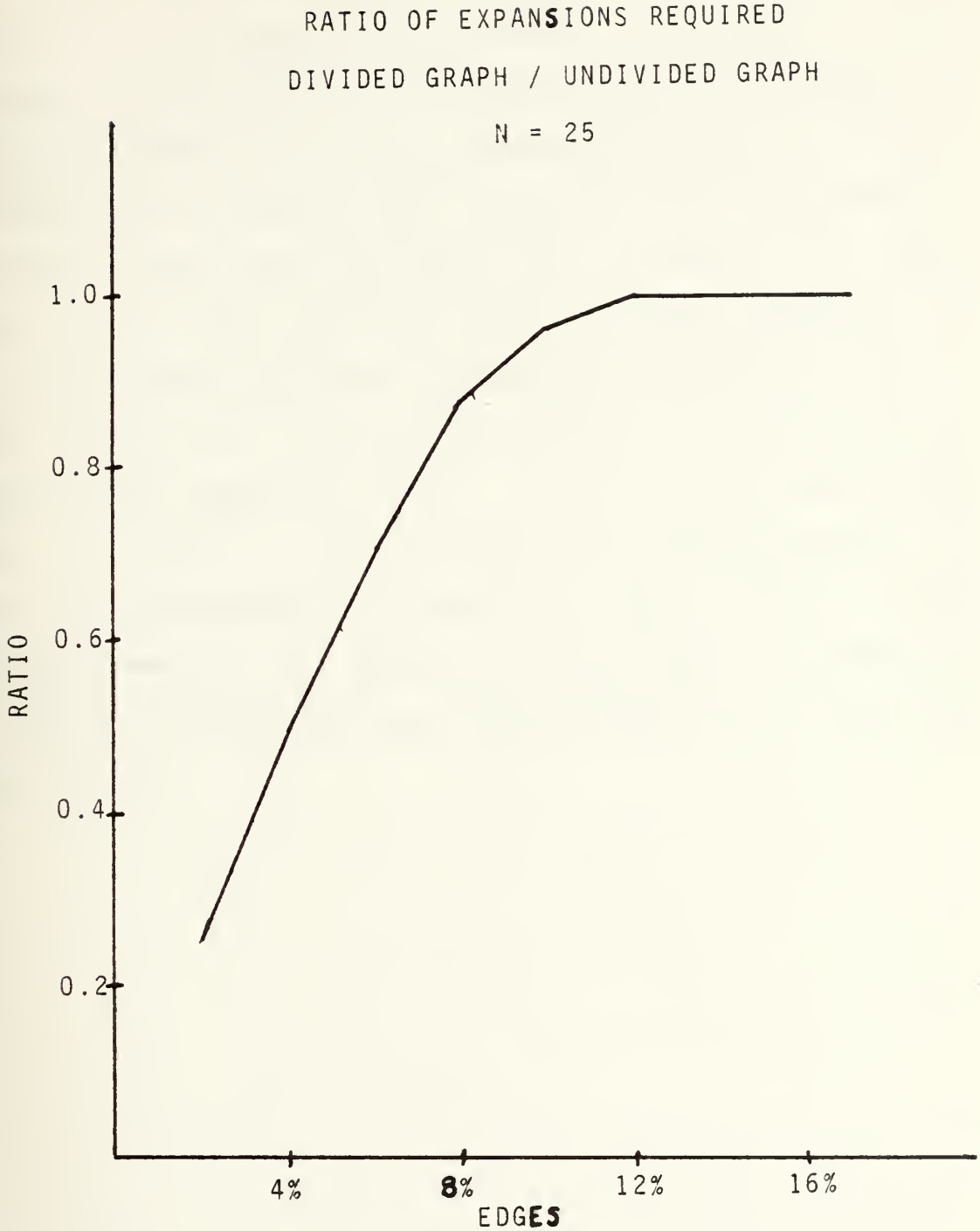

RATIO OF EXPANSIONS REQUIRED
DIVIDED GRAPH / UNDIVIDED GRAPH
N = 25
1.0
0.8
0.6
0.4
0.2
RATIO
4%
8%
12%
16%
EDGES

Figure 18

scheduling problem in a large university. The vertices of the graph are the classes and there is a edge between two vertices if at least one student is enrolled in both classes. The question then is: How many exam periods are needed to schedule all final examinations with no conflicts? Consider the hypothetical but realistic case of a university with 200 classes, and a average of 20 students per class. Since many students take the same curriculum together it is not inconceivable to have a graph with an average degree of each vertex of 5. This would result in a graph of 200 vertices and 500 edges or 2 1/2 %. This is a very sparse graph which is likely to have several articulation points. In fact, the biconnected components are likely to be the various departments of the university. Dividing the graph prior to solving this problem would surely result in a time and space savings.

IV. CONCLUSIONS

The branch and bound algorithm described in this paper can be used to find the chromatic number of graphs. Though no empirical evidence was gained for graphs larger that 32 vertices, it is felt that somewhat larger graphs can also be colored in a reasonable amount of time on the average with the LEAST COLOR AVAILABLE method of ordering the vertices. Further, some improvement in the lower bound function or search strategy might be possible which would make the algorithm more efficient. For example, some lookahead procedure might conceivably improve the method of assigning costs to each node of the search tree. Also, the priority of a node in the queue was primarily the number of colors in the partial coloring and only secondarily a function of the depth in the tree. A possible improvement is to make the priority some algebraic combination of the two factors thus giving more emphasis on the depth in the tree. This might result in some faster colorings. Since the emphasis here was on finding the best branching function, little effort was expended in looking for improvement in the other 2 components of the branch and bound strategy.

For the five branching functions investigated, LCA seems far superior for all graphs in both the time required and the amount of computer storage space needed. Though all

five functions have worst case time and space complexity which appears to be exponential, the LCA method has an average case time complexity which appears to be nearly linear in the range of graphs from 5 to 32 vertices.

Though the research and experiments done here focused on the exact algorithm for graph coloring, a few very minor changes would result in an approximation algorithm. The function EXPAND could be changed such that it only generates the first possible child node rather than all children of a node. No priority queue would be needed and the number of nodes required in the search tree would just be n, the number of vertices in the graph. So the space-complexity of the algorithm would be linear in n. The time required would be $O(n^4)$ if LCA were used for an ordering function, but it would be interesting to see how good its ability to approximate the real chromatic number is compared to other orderings.

Dividing a graph at its articulation points prior to coloring is an effective way to reduce the time required to find the chromatic number. However, unless the graph is known to have articulation points or unless it is very sparse, the effort spent in finding the biconnected components may not be worth it. When a graph does have articulation points, there is a two-fold benefit gained by divid-

ing the graph. The first is, of course, that the size of the resulting graph is smaller and thus easier to color. The second benefit comes from the fact that the biconnected components tend to be dense which makes coloring easier.

APPENDIX A

This appendix contains the source listing for the computer programs used in this research. The program is organized into various functions which share some common global variables. These variables are listed in the module EXTERN DECS. The remainder of the modules contain one or more functions grouped together according to their logical function. The modules and the function in this appendix are:

MODULE	FUNCTIONS
EXTERN DECS	none
MAIN	main()
GENGRAPH	gengraf()
	random()
ORDERNODES	ordernode()
	picknode()
COLORALL	coloral1()
	expand()
	convb()
DIVIDE	divide()
	proc()
TREEMANAGER	getnode()
	addq()
	gettop()

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                    GLOBAL MACRO DEFINITIONS
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
#define NUMNODES   4000    /* Maximum number of tree nodes */
#define NUMQ       2500    /* Maximum size of the priority Q */
#define NULL          0    /* The empty pointer */
#define NUMTRIALS   100    /* The number of trials for each size
                              of graph */
#define MAXSIZE      30    /* The largest graph size (n) */
#define MINSIZE      10    /* The smallest graph */


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                     EXTERNAL DECLARATIONS
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
                /* trnode is the structure of the nodes
                which make up the search tree */
 struct trnode {
    unsigned depth :8;
    unsigned cost :8;
    unsigned vertex :8;
    unsigned color :8;
    struct trnode *parptr;
    };
                /* tnode is a array of search tree nodes.
                q is an array of pointers to tree nodes */
struct trnode tnode[NUMNODES],*q[NUMQ];
                /* ndata is a record of data regarding the
                graph vertices to be used in ordering the
                vertices by the branching function.
                order is an array of records */
struct ndata {
    unsigned next   : 5;
    unsigned degree   : 5;
    unsigned flag     : 1;
    } order[32];
int fd;         /* file descriptor for output */
int n;          /* number of vertices */
int count;      /* number of calls to EXPAND */
int hinode;     /* first vertex to color. */
int index;      /* index of next available node in the
                array of tree nodes - also the number
                of nodes in the search tree at any time */
int qsize;      /* index into the priority queue */
int chrom;      /* The chromatic number of the graph */
int color[32];  /* The color class of each vertex */
int ordnum;     /* The ordering function being used */
int saven;      /* The saved value of n for DIVIDE */
```

```
int graf[32][32];  /* The adjacency matrix */
float numsubs;  /* The number of biconnected subgraphs */
float bigsubn;  /* The size of the largest biconn. comp. */
float bigsube   /* Number of edges in bigsubn */
long int graph[32];  /* The adjacency matrix A in bit strings */
int sonct[32];  /* A counter of child nodes for each level */
int parct[32];  /* Parent counter for each level */


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                              MAIN
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
                /* This function provides for the flow of
                control throughout the program and records
                the data which is to be collected.  It also
                contains the code for any output generated */
main () {

int full;       /* The percentage of edges */
int edges;      /* The number of edges */
int i,j;        /* iteration variables */
int ndflag[6];  /* Too large of a search tree flag for each
                   ordering method */
int trial;      /* The trial number for each graph size */
float maxedges; /* The maximum edges possible for graph */
float maxbr[6]; /* The maximum expansions required */
float maxnd[6]; /* The max nodes required */
float avgbr[6]; /* The average expansions required */
float avgnd[6]; /* The average nodes in the search tree */
char *label[6];

                /* The output file and labels */
fd = copen("/work/rerauten/out3",'a');
label[1] = "RANDOM";
label[2] = "DEGREE";
label[3] = "DEG  3";
label[4] = "ADJCEN";
label[5] = "LCA";

                /* For each value of n desired */
for(n=MINSIZE;n<=MAXSIZE;n+=5) {

                /* compute maximum edges possible */
    maxedges = (n * (n-1))/2;

                /* For each value of the percentage of
                   edges desired */
    for(full=2;full<=20;full+=2) {
```

```
        /* compute number of edges */
edges = full * maxedges/100;

        /* Print headings */
printf(fd,"0 =%3d, EDGES =%4d(%2d%%)0,
        n,edges,full);
printf(fd,"%5s%10s%10s0,"order","branches","nodes");
printf(fd,"%15s%12s","avg  max","avg  max ");
printf(fd,"0);

        /* Initialize data collection variables */
for(i=1;i<=5;i++) {
    maxbr[i] = 0;
    maxnd[i] = 0;
    avgbr[i] = 0;
    avgnd[i] = 0;
    ndflag[i] = 0;
    }
        /* For the number of trials desired */
for(trial=1;trial<=NUMTRIALS;trial++) {

        /* Generate a random graph */
    gengraf(n,edges);

        /* For each ordering method */
    for(ordnum=5;ordnum<=5;ordnum++) {
        if(! ndflag[ordnum]) {

        /* Divide the graph and color the components */
            if(! divide()) ndflag[ordnum] = 1;

        /* Record data */
            avgbr[ordnum] += count;
            avgnd[ordnum] += index;
            if(count > maxbr[ordnum])
                maxbr[ordnum] = count;
            if(index > maxnd[ordnum])
                maxnd[ordnum] = index;
            }
        }
    }
        /* Print out the data collected */
for(i=1;i<=5;i++) {
    if(! ndflag[i]) {

        /* Compute the averages */
        avgbr[i] /= NUMTRIALS;
        avgnd[i] /= NUMTRIALS;
```

```
                printf(fd,"%6s",label[i]);
                printf(fd,"%4.0f%5.0f",avgbr[i],maxbr[i]);
                printf(fd,"%6.0f%5.0f  ",avgnd[i],maxnd[i]);
                printf(fd,"0);
                }
            else printf(fd,"%6s  NOT ENOUGH SPACE AVAIL0,
                        label[i]);
            }
        }
    }
cexit();

}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                        GENERATE GRAPH
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
            /* This function generates a random graph with
                n vertices and e edges and records it in a
                global adjacency matrix */

gengraf(n,e) int n,e; {

int i,j,a,b;
long int convb();
            /* initialize the adjacency matrix */
for (i=0;i<=n;i++) {
    graph[i] = 0;
    for(j=0;j<=n;j++) graf[i][j] = 0;
    graf[i][i] = 1;
    }
while (e > 0) {
            /* Get 2 random vertices */
    a = random(n);
    b = random(n);
            /* Find the next ungenerated edge */
    while(graf[a][b] == 1) {
        a = (a % n) + 1;
        if (a == 1) b = (b % n) + 1;
        }
            /* Record the new edge */
    graf[a][b] = 1;
    graf[b][a] = 1;
    e--;
    }
            /* Put adj matrix in array of bit strings */
for (i=1;i<=n;i++) {
```

```
    graf[i][i] = 0;
    for(j=1;j<=n;j++) if(graf[i][j]) graph[i] |= convb(j);
    }
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                          RANDOM
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
            /* This function is used to get random numbers
               between 1 and the input argument n.
               taken from Grogono [ref. --] */
int random(n) int n; {

static long int seed 18403;
int i;

seed = (25173 * seed + 13849) % 65536;
i = seed % n;
return(i+1);
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                         ORDERNODE
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
               /* This function orders the vertices of
               the graph for each of the static ordering
               methods.  It does so by filling the array
               order.next with the vertex to follow each
               vertex in the graph. */
ordernode()  {

int i,j,k;       /* iteration variables */
int save;        /* the saved last vertex */
int temphi;      /* the temporary next vertex */
int max;         /* any maximum needed to maintain */
int colsum[32]; /* used in Adjacency ordering to keep
               track of number of previously colored
               vertices a vertex is adjacent to. */
int maxcol;      /* The column with the max 1's in colsum */
int hideg;       /* The highest degree of any vertex */
float dvec[4][32];  /* The vector of degrees in DVEC 3 */
float maxdvec;  /* The max vector calculated in DVEC 3 */
long int mask;  /* A bit manipulation mask */
long int convb(); /* A converting function (SUPP MOD) */

max = 0;
```

```
                /* initialize the vertex data */
for(i = 0;i<=n;i++) {
         order[i].degree = 0;
         order[i].next = 0;
         order[i].flag = 0;
         mask = graph[i];

                /* Compute the degree of each vertex */
         for(j=1;j<=n;j++) {
              if ((mask & 01) == 1) order[i].degree += 1;
              mask >>= 1;
              }

                /* find the max degree and the vertex */
         if(order[i].degree > max) {
                max = order[i].degree;
                hinode = i;
                }
         }

                /* Order the vertices depending on the
                current ordering number */
switch (ordnum) {
    case 0:
            break;

                /* RANDOM */
    case 1:
        for(i=0;i<n;i++) order[i].next = i + 1;
        hinode = 1;
        break;

                /* by DEGREE */
    case 2:
        order[0].next = hinode;

                /* .flag indicates already colored vertex */
        order[hinode].flag = 1;
        save = hinode;

                /* For each vertex find the next one */
        for(i=1;i<=n;i++) {
            max = 0;

                /* Find the highest degree of the uncolored
                    vertices */
            for(j=1;j<=n;j++) {
                if((order[j].degree > max)&&(! order[j].flag)) {
```

```
                max = order[j].degree;
                temphi = j;
                }
            }

            /* record the next vertex */
        order[save].next = temphi;
        order[temphi].flag = 1;
        save = temphi;
        }
        break;

            /* by DVEC 3 */
case 3:

            /* initialize 0th iteration to degree */
    for(i=1;i<=n;i++) dvec[0][i] = order[i].degree;

            /* For 3 iterations */
    for(i=1;i<=3;i++) {

            /* For each vertex */
        for(j=1;j<=n;j++) {
            dvec[i][j] = 0;

            /* For every other vertex in the graph */
            for(k=1;k<=n;k++) {
                mask = convb(k);

            /* If they are connected compute summation
            for the ith iteration */
                if((graph[j] & mask) > 0)
                    dvec[i][j] += dvec[i-1][k];
                }
            }
        }
            /* Order the vertices */
            /* The ordering method is identical to case 2
            except for the ordering parameter */
    save = 0;
    for(i=1;i<=n;i++) {
        maxdvec = 0;
        for(j=1;j<=n;j++) {
            if((dvec[3][j] > maxdvec) && (! order[j].flag)) {
                maxdvec = dvec[3][j];
                temphi = j;
                }
            }
```

```
        order[save].next = temphi;
        order[temphi].flag = 1;
        save = temphi;
        }
    hinode = order[0].next;
        break;

            /* by ADJACENCY */
case 4:
            /* initialize */
    for(i=1;i<=n;i++) colsum[i] = 0;
    order[0].next = hinode;
    order[hinode].flag = 1;
    save = hinode;
    colsum[save] -= 33;

            /* For each vertex find the next one */
for(j=1;j<=n-1;j++) {
    max = 0;
    mask = 01;
            /* increment the bit string colsum with each
            vertex added to the set of colored vertices */
    for(i=1;i<=n;i++)  {
        if((graph[save] & mask) > 0)
            colsum[i]++;

            if(colsum[i] > max)max = colsum[i];
        mask <<= 1;
        }

            /* Order the vertices - again the same
            ordering method is used as in case 2 except
            the ordering parameter is by value of colsum */
    hideg = 0;
    for(i=1;i<=n;i++) {
        if((colsum[i]==max)&&(order[i].degree>hideg)
            &&(! order[i].flag)) {
            hideg = order[i].degree;
            temphi = i;
            }
        }
        order[save].next = temphi;
        order[temphi].flag = 1;
        colsum[temphi] -= 33;
        save = temphi;
    }
        break;
```

```
                /* by LEAST COLORS AVAILABLE */
        case 5:
                /* no work done in this function.
                See PICKNODE */
            break;
    }
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                        PICKNODE
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                This function is called from the routime
                EXPAND to get the next vertex to color.
                If one of the static orderings is being
                used it just looks up the next vertex in
                the table order.next.  If the dynamic
                ordering LCA is being used, it must trace
                the search tree to the root to determine
                the next vertex to color.  */

picknode(nodeptr) struct trnode *nodeptr; {

                /* Declarations */
long int cc[32],vdone,longl;
int max,isum,k,i,sum[32],nodenum,hideg;
struct trnode *tempptr;

                /* If static ordering, just table lookup */
if(ordnum < 5) return(order[nodeptr->vertex].next);

                /* initialize
                vdone is a bit string to record which
                vertices have already been colored.
                cc is the color class matrix
                sum counts the colors not avail for each
                vertex. */
vdone = 0;
for(i=1;i<=n;i++) {
    cc[i] = 0;
    sum[i] =0;
    }
tempptr = nodeptr;

                /* Trace from node N to root. */
while (tempptr != NULL) {

                /* fill color class matrix */
```

```
    cc[tempptr->color] |= graph[tempptr->vertex];

                /* record colored vertices */
    vdone |= convb(tempptr->vertex);
    tempptr = tempptr->parptr;
    }
longl = 01;
max = 0;

                /* For each of the n vertices */
for(k=1;k<=n;k++) {

                /* If the vertex  k is not colored */
    if((vdone & convb(k)) == 0) {
        isum = 0;

                /* count the color classes to which vertex
                k cannot be added */
        for(i=1;i<=nodeptr->cost;i++) {
            isum += longl & (cc[i]>>(k-1));
            }
        sum[k] = isum;

                /* keep track of fewest number of color
                classes available */
        if(isum > max) max = isum;
        }
    }
hideg = 0;

                /* Of all the vertices with the fewest color
                classes available, find the one with the
                highest degree */
for(i=1;i<=n;i++) {
    if((sum[i] == max) && (!(convb(i) & vdone))) {
        if(order[i].degree > hideg) {
            hideg = order[i].degree;
            nodenum = i;
            }
        }
    }
                /* Return the vertex number of the next
                vertex to color. */
return(nodenum);
}
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                        COLOR ALL
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
                This function creates the root node of the
                search tree, puts it in the priority queue
                then calls the function EXPAND to expand
                the top node in the queue until the top of
                the queue is a complete coloring. */
coloral1() {

struct trnode *nodeptr,*gettop();
int i;

                /* Initialize for new coloring */
k = 0;
index = 0;
qsize = 0;
count = 0;
tnode[0].cost = 31;
for (i=0;i<NUMQ;i++) q[i] = &tnode[0];

                /* Get an empty node, fill it with the
                data for the root of the tree and
                put it in the queue. */
if (! (nodeptr = getnode()))  return(0);
nodeptr->parptr = NULL;
nodeptr->depth = 1;
nodeptr->cost = 1;
nodeptr->vertex = hinode;
nodeptr->color = 1;
addq(nodeptr);
                /* 'forever' */
for(;;) {
                /* Get the top node in the queue */
    nodeptr = gettop();

                /* If this is complete coloring, quit */
    if(nodeptr->depth == n) break;
    else if (! expand(nodeptr)) return(0);
    }

                /* Record the chromatic number */
chrom = nodeptr->cost;

                /* Record the color class assignment of
                each vertex in this path. */
while (nodeptr->parptr != NULL) {
    color[nodeptr->vertex] = nodeptr->color;
    nodeptr = nodeptr->parptr;
```

```
    }
color[nodeptr->vertex] = nodeptr->color;
return(1);
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                          EXPAND
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                This function expands the node pointed to by
                the variable nodeptr.  It creates a child
                node for each possible color available as
                well as a node for one new color.  */

expand (nodeptr) struct trnode *nodeptr; {

struct trnode *tempptr,*newnode,*getnode();
int vertexnum,i;
long int mask,colavail,convb();

                /* increment counters */
count += 1;
parct[nodeptr->depth]++;

                /* Get the number of the next vertex to color */
vertexnum = picknode(nodeptr);

                /* create new nodes for all colorings using
                only existing colors */
colavail = 01;
for (i=1;i<nodeptr->cost;i++) {
    colavail <<= 1;
    colavail |= 01;

                /* colavail is a bit string with a one in the
                position of every available color. */
    }
tempptr = nodeptr;

                /* For each node N in the path to the root */
for (;;) {

                /* If N.vertex and vertexnum are connected */
    if((graph[vertexnum] & convb(tempptr->vertex)) != 0) {
        mask = convb(tempptr->color);

                /* Remove N.color from colavail */
```

```
        if ((colavail & mask) != 0) colavail ^= mask;
        }
    if (tempptr->parptr == NULL) break;
    tempptr = tempptr->parptr;
    }
                /* If any colors are left */
if (colavail != 0) {
    mask = 01;

                /* For each color available */
    for (i=1;i<=nodeptr->cost;i++) {
        if ((colavail & mask) != 0) {

                /* Generate child node */
            if ((newnode = getnode()) == 0) return(0);
            newnode->depth = nodeptr->depth + 1;
            newnode->cost = nodeptr->cost;
            newnode->vertex = vertexnum;
            newnode->color = i;
            newnode->parptr = nodeptr;
                /* increment child counter */
            sonct[nodeptr->depth]++;

                /* Put the node in the queue */
            addq(newnode);
            }
        mask =<< 1;
        }
    }
                /* Generate one more node with a new color */
if ((newnode = getnode()) == 0) return(0);
newnode->depth = nodeptr->depth + 1;
newnode->cost = nodeptr->cost + 1;
newnode->vertex = vertexnum;
newnode->color = nodeptr->cost + 1;
newnode->parptr = nodeptr;
sonct[nodeptr->depth]++;
addq(newnode);
return(1);
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                        CONVERT TO BIT
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
                /* This function converts an integer argument
                between 0 and 16 to a bit string which has a
                1 in the position of the argument and 0's
```

```
                everywhere else */
long int convb(arg) int arg; {

int i;
long int bitpos;

bitpos = 01;
for(i=1;i<arg;i++) bitpos =<< 1;
return(bitpos);
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                          DIVIDE
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
                /* This function divides the graph into its
                biconnected components.  It then creates a
                new adjacency matrix for each component and
                calls the function COLORALL to color the
                component.  It records the data for each
                component and sums it up for all comps of a
                graph. When it returns, the sum data is in
                the variables set up by MAIN. */

divide() {

int number[32];    /* to record the visiting order of vertices*/
int back[32];      /* record the lowest vertex to which a back
                      edge exist */
int stackv[32];    /* A stack of the vertices  */
int num;           /* To maintain the next number */
int w,v,top;       /* vertices - top is top of stack */
int stvptr;        /* pointer to top of vertex stack */
int steptr;        /* pointer to top of edge stack */
int nume;          /* number of edges in bi-component */
int numv;          /* number of vertices in bi-component */
int vnum[32];      /* array to record the vertex numbers as
                     the edges of the edge stack are popped */
int totcount;      /* count the total calls to EXPAND for all
                      components */
int totindex;      /* count the total nodes used */
int table[32];     /* to convert the bicomponent to a graph */
int tableptr;      /* pointer into table */
int vert1,vert2;   /* the vertices of the edges in new graph */
long int convb();
int i,j;
                /* An edge in a biconnected component defined
                by the two vertices at its end */
```

```
struct edge {
    unsigned v1: 8;
    unsigned v2: 8;
    } stacke[350];
                /* Stacke is an stack of edges */

                /* Save off the oringginal value of n - the
                biconnected components will be different */
saven = n;

                /* Initialize all variables */
number[1] = 1;
num = 2;
stackv[0] = 1;
top = 1;
for(v=2;v<=n;v++) number[v] = 0;

steptr = -1;
stvptr = 0;
totcount = 0;
totindex = 0;

                /* Begin a depth first search of the graph
                exploring forward edges */
forward:
                /* While there is an unexplored forward edge
                from the top vertex on the stack to some
                vertex w */
while((w = proc(top)) != 0) {
                /* Stack the edge (top,w) on the edge stack */
    stacke[++steptr].v1 = top;
    stacke[steptr].v2 = w;
                /* If w has already been explored give the min
                of number(w) and back(top) to back(top). */
    if(number[w] > 0) {
        if(number[w] < back[top]) back[top] = number[w];
        }
                /* Else stack w and explore it */
    else {
        number[w] = num++;
        back[w] = number[w];
        stackv[++stvptr] = w;
        top = w;
        }
    }

                /* If there is no forward edge then backup
                to the last vertex which still has a forward
```

```
                edge which is unexplored */
backup:
if(stvptr > 0) {
                /* try the next to top vertex */
    v = stackv[stvptr - 1];
                /* if it has a number less that the back of
                the top vertex then we have an articulation
                point and the bi connected component is on
                top of the edge stack */
    if(back[top] >= number[v]) {
                /* increment and initialize counters */
        numsubs += 1;
        nume = 0;
        numv = 0;
        for(i=1;i<=saven;i++) graph[i] = 0;
        tableptr = 0;
        table[0] = 0;
        for(i=0;i<=saven;i++) vnum[i] = 0;
                /* pop edges until end of component */
        while((number[stacke[steptr].v1] >= number[v]) &&
            (number[stacke[steptr].v2] >= number[v])
            && (steptr >= 0)) {

                /* increment data counters */
            nume++;
            if(++vnum[stacke[steptr].v1] == 1) numv++ ;
            if(++vnum[stacke[steptr].v2] == 1) numv++ ;
            vert1 = 0;
            vert2 = 0;
                /* check to see if either vertex is already
                in the table - if so, get the index which
                is going to be the new vertex number */
            for(i=0;i<=tableptr;i++) {
                if(table[i] == stacke[steptr].v1) vert1 = i;
                if(table[i] == stacke[steptr].v2) vert2 = i;
                }
                /* if not in table, put it there and get new
                vertex number */
            if(vert1 == 0) {
                tableptr++;
                table[tableptr] = stacke[steptr].v1;
                vert1 = tableptr;
                }
            if(vert2 == 0) {
                tableptr++;
                table[tableptr] = stacke[steptr].v2;
                vert2 = tableptr;
                }
```

```
                /* record the edge between the new vertex
                numbers */
            graph[vert1] |= convb(vert2);
            graph[vert2] |= convb(vert1);
                /* decrement edge stack pointer and do again */
            steptr--;
            }
                /* The biconnected component is now recorded
                in the adjacency matrix and ready to color */
                /* Is this the biggest component of this graph */
        if(numv > bigsubn) bigsubn = numv;
        if(nume > bigsube) bigsube = nume;
                /* new n is size of the table */
        n = tableptr;
                /* Call the ordering function */
        ordernode();
                /* Color the component */
        if(! coloral1()) return(0);
                /* Increment the cummulative counters */
        totcount += count;
        totindex += index;
        }
                /* If no art. pt. pop the vertex stack and
                explore any forward edges of the vertex
                which is now on top */
    else if(back[top] < back[v]) back[v] = back[top];
    top = stackv[--stvptr];
    goto forward;
    }

                /* When all the connected vertices have
                been explored, check for another unconnected
                component and explore it */
for(i=2;i<=saven;i++) {
    if(number[i] == 0) {
        number[i] = 1;
        num = 2;
        stackv[0] = i;
        top = i;
        stvptr = 0;
        steptr = -1;
        goto forward;
        }
    }
                /* When everything is done and all components
                are found and colored, set the counters to the
                cummulative totals, reset n, and reset the
                adjacency matrix */
```

```
count = totcount;
index = totindex;
n = saven;
for (i=1;i<=n;i++) {
    graf[i][i] = 0;
    for (j=1;j<=n;j++) {
        if (graf[i][j] == 2) graf[i][j] = 1;
        if (graf[i][j] == 1) graph[i] |= convb(j);
        }
    }
return(1);
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                        PROC
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
                /* This function determines if we can
                proceed in the depth first search exploring
                forward edges.  It marks the adjacency
                matrix with a 2 as each edge is explored
                so that no edge is explored twice */
proc(i) int i; {

int j;

                /* Is there a vertex adjacent to vertex i
                on an edge which hasn't been explored. If so
                mark it explored and return the vertex number */
for(j=1;j<=saven;j++)
    if(graf[i][j] == 1) {
        graf[i][j] = 2;
        graf[j][i] = 2;
        return(j);
        }
return(0);
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                        TREE MANAGER
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
                /* This module contains the functions
                necessary to manage the search tree nodes
                as well as the priority queue. */
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                              GETNODE
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
                /* This functions maintains the array of
                nodes and returns a pointer to the next
                available node when called.  Index is a
                global variable */
struct trnode *getnode() {
                /* index gets initialized in COLORALL */
index += 1;
if (index < NUMNODES) return(&tnode[index]);
else return(0);
}


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                              ADD TO QUEUE
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
                /* This functions adds a new node to the
                priority queue. It puts it in the last
                position and then 'sifts it up' until it is
                in its proper location */
addq (ptr) struct trnode *ptr; {

struct trnode *temp;
int i,j;
                /* Put the new node at the bottom */
qsize += 1;
if (qsize == NUMQ) printf(fd,"Q OVERFLOW0);
q[qsize] = ptr;
                /* j is the node being sifted up */
j = qsize;
                /* i is the parent of j */
i = j/2;
                /* While j is not the root */
while(j > 1) {
                /* compare priority of j to i */
    if(q[i]->cost < q[j]->cost) return;
    else if (q[i]->cost == q[j]->cost) {
        if (q[i]->depth >= q[j]->depth) return;
        }
                /* Exchange nodes and continue */
    temp = q[j];
    q[j] = q[i];
    q[i] = temp;
    j = i;
    i = j/2;
    }
}
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                            GET TOP
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

                /* This functions returns a pointer to the
                tree node which is on top of the priority
                queue.  The queue is a heap.  When the root
                of the full binary tree is removed, the node
                in the last position is moved to the root and
                then 'sifted down' */
struct trnode *gettop() {

struct trnode *saveptr,*temp;
int i,j;
                /* save off the root node */
saveptr = q[1];
                /* move the last node to the root */
q[1] = q[qsize];
                /* decrement the qsize */
qsize -= 1;
                /* sift the root node down */
                /* i is the parent node being looked at */
i = 1;
                /* j is the left child of i */
j = 2 * i;
                /* While were not out of the heap */
while(j <= qsize) {
    if (j < qsize) {

                /* Get the child of i with least cost */
        if (q[j]->cost > q[j+1]->cost) j = j+1;

                /* if both the same cost, get greater depth */
        else if (q[j]->cost == q[j+1]->cost)
            if(q[j]->depth < q[j+1]->depth) j = j+1;
        }
                /* If both children have lower priority, quit */
    if (q[i]->cost < q[j]->cost) break;
    else if (q[i]->cost == q[j]->cost)
        if (q[i]->depth >= q[j]->depth) break;

                /* exchange node i with j(its highest
                priority child) */
    temp = q[j];
    q[j] = q[i];
    q[i] = temp;
                /* set i = to j and do it again */
    i = j;
```

```
    j = 2 * i;
    }
                /* when done, return the save off root */

return(saveptr);
}
```

APPENDIX B

This appendix contains output listings from the various programs run to generate the plots described in Section IV. The first listing is the data for graph coloring without dividing for graphs with from 10 to 30 vertices. The edge values are 30%, 50%, and 70%.

The second listing is the output from the DIVIDE routine which divides the graphs and records data regarding the size of the biconnected components and the number of components. The graphs ranged in size from 6 to 100 vertices.

The next listing contains data from the routine to divide the graphs and color the components. Graphs with 25 vertices and 2% to 14% edges were used in this experiment.

The last listing shows the effects of the amount of edges when n is held constant. For this experiment n = 20 while the edges vary from 20% to 80% by 5% intervals.

* *
COMPARISON OF 5 ORDERING METHODS FOR SEVERAL GRAPH SIZES
* *

N = 10, EDGES = 13(30%)

ORDER	BRANCHES		NODES	
	avg	max	avg	max
RANDOM	19	65	38	115
DEGREE	10	14	21	29
DVEC 3	9	11	21	25
ADJCEN	9	9	21	24
LCA	9	9	21	24

BRANCH FACTOR BY LEVEL

	1	2	3	4	5	6	7	8	9	10
RANDOM	1.55	1.92	1.96	2.14	1.75	2.11	1.85	2.43	2.01	0.00
DEGREE	1.23	1.59	1.82	1.94	2.29	2.41	2.57	2.78	2.88	0.00
DVEC 3	1.04	1.45	1.96	2.22	2.45	2.47	2.70	2.83	2.87	0.00
ADJCEN	1.00	1.29	2.16	2.44	2.41	2.45	2.68	2.83	2.89	0.00
LCA	1.00	1.29	2.15	2.29	2.47	2.61	2.74	2.87	2.92	0.00

N = 10, EDGES = 22(50%)

ORDER	BRANCHES		NODES	
	avg	max	avg	max
RANDOM	29	129	53	239
DEGREE	10	18	21	35
DVEC 3	9	11	20	25
ADJCEN	9	11	20	25
LCA	9	9	20	25

BRANCH FACTOR BY LEVEL

	1	2	3	4	5	6	7	8	9	10
RANDOM	1.34	1.81	1.72	1.99	1.63	2.24	1.57	2.13	1.75	0.00
DEGREE	1.08	1.36	1.59	1.76	1.93	2.25	2.57	2.66	3.13	0.00
DVEC 3	1.01	1.18	1.42	1.97	2.27	2.42	2.56	2.73	3.13	0.00
ADJCEN	1.00	1.00	1.35	2.13	2.44	2.50	2.65	2.80	3.16	0.00
LCA	1.00	1.00	1.35	2.07	2.35	2.45	2.72	2.87	3.24	0.00

N = 10, EDGES = 31(70%)

ORDER	BRANCHES		NODES	
	avg	max	avg	max
RANDOM	33	160	56	250
DEGREE	10	17	19	27
DVEC 3	9	16	18	26
ADJCEN	9	10	18	22

```
   LCA    9     9     18    22

                BRANCH FACTOR BY LEVEL
         1     2     3     4     5     6     7     8     9    10
RANDOM 1.08  1.50  1.48  1.98  1.51  2.07  1.43  2.33  1.48  0.00
DEGREE 1.01  1.12  1.24  1.36  1.58  1.98  2.31  2.63  3.10  0.00
DVEC 3 1.02  1.05  1.19  1.21  1.61  2.15  2.51  2.73  3.14  0.00
ADJCEN 1.00  1.00  1.00  1.13  1.78  2.41  2.53  2.72  3.14  0.00
   LCA 1.00  1.00  1.00  1.13  1.78  2.23  2.52  2.76  3.25  0.00


N = 15, EDGES =  31(30%)

ORDER  BRANCHES      NODES
       avg  max    avg  max
RANDOM  NOT ENOUGH SPACE AVAIL
DEGREE  22    88     48  173
DVEC 3  17    42     38   82
ADJCEN  15    34     36   66
   LCA  14    22     35   44

                BRANCH FACTOR BY LEVEL
         1     2     3     4     5     6     7     8     9    10
RANDOM   NOT CALCULATED
DEGREE 1.21  1.59  1.67  1.86  2.03  2.03  2.09  2.22  2.36  2.49
DVEC 3 1.06  1.24  1.77  2.07  2.11  2.28  2.31  2.35  2.53  2.73
ADJCEN 1.00  1.02  1.64  2.13  2.45  2.48  2.47  2.46  2.49  2.65
   LCA 1.00  1.02  1.62  2.07  2.36  2.41  2.44  2.57  2.59  2.71

         11    12    13    14    15
RANDOM   NOT CALCULATED
DEGREE 2.66  2.83  3.05  3.44  0.00  0.00  0.00  0.00  0.00  0.00
DVEC 3 2.77  2.88  3.20  3.43  0.00  0.00  0.00  0.00  0.00  0.00
ADJCEN 2.78  2.98  3.15  3.44  0.00  0.00  0.00  0.00  0.00  0.00
   LCA 2.82  3.07  3.32  3.60  0.00  0.00  0.00  0.00  0.00  0.00


N = 15, EDGES =  52(50%)

ORDER  BRANCHES      NODES
       avg  max    avg  max

RANDOM  NOT ENOUGH SPACE AVAIL
DEGREE  24    72     47  129
DVEC 3  19    47     38   89
ADJCEN  16    61     35  115
   LCA  15    33     32   66
```

BRANCH FACTOR BY LEVEL

	1	2	3	4	5	6	7	8	9	10
RANDOM	NOT CALCULATED									
DEGREE	1.14	1.30	1.47	1.57	1.72	1.81	1.85	2.07	2.30	2.25
DVEC 3	1.04	1.17	1.34	1.59	1.98	1.96	2.04	2.21	2.22	2.45
ADJCEN	1.00	1.00	1.04	1.49	2.02	2.43	2.39	2.41	2.25	2.41
LCA	1.00	1.00	1.04	1.47	1.95	2.21	2.21	2.29	2.37	2.50

	11	12	13	14	15					
RANDOM	NOT CALCULATED									
DEGREE	2.32	2.57	2.71	3.14	0.00	0.00	0.00	0.00	0.00	0.00
DVEC 3	2.49	2.55	2.70	3.16	0.00	0.00	0.00	0.00	0.00	0.00
ADJCEN	2.53	2.63	2.87	3.22	0.00	0.00	0.00	0.00	0.00	0.00
LCA	2.54	2.70	2.95	3.38	0.00	0.00	0.00	0.00	0.00	0.00

N = 15, EDGES = 73(70%)

ORDER	BRANCHES		NODES	
	avg	max	avg	max
RANDOM	NOT ENOUGH SPACE AVAIL			
DEGREE	24	75	44	128
DVEC 3	21	56	38	96
ADJCEN	15	32	31	66
LCA	14	28	29	55

BRANCH FACTOR BY LEVEL

	1	2	3	4	5	6	7	8	9	10
RANDOM	NOT CALCULATED									
DEGREE	1.02	1.07	1.15	1.31	1.42	1.60	1.71	1.78	1.98	2.20
DVEC 3	1.02	1.02	1.06	1.24	1.36	1.60	1.77	2.01	2.05	2.33
ADJCEN	1.00	1.00	1.00	1.00	1.08	1.49	2.03	2.31	2.40	2.49
LCA	1.00	1.00	1.00	1.00	1.08	1.47	1.93	2.14	2.14	2.35

	11	12	13	14	15					
RANDOM	NOT CALCULATED									
DEGREE	2.21	2.52	2.70	3.18	0.00	0.00	0.00	0.00	0.00	0.00
DVEC 3	2.27	2.46	2.68	3.15	0.00	0.00	0.00	0.00	0.00	0.00
ADJCEN	2.45	2.54	2.85	3.14	0.00	0.00	0.00	0.00	0.00	0.00
LCA	2.49	2.63	2.91	3.44	0.00	0.00	0.00	0.00	0.00	0.00

N = 20, EDGES = 57(30%)

ORDER	BRANCHES		NODES	
	avg	max	avg	max

RANDOM	NOT ENOUGH SPACE AVAIL			
DEGREE	63	312	125	601
DVEC 3	33	118	70	229
ADJCEN	25	208	57	418
LCA	22	121	53	247

BRANCH FACTOR BY LEVEL

	1	2	3	4	5	6	7	8	9	10
RANDOM	NOT CALCULATED									
DEGREE	1.53	1.59	1.72	1.82	1.86	1.60	1.76	2.08	2.08	2.11
DVEC 3	1.25	1.34	1.63	1.87	2.01	2.05	2.06	2.28	2.41	2.32
ADJCEN	1.00	1.00	1.49	1.84	2.23	2.22	2.42	2.40	2.70	2.76
LCA	1.00	1.00	1.49	1.77	2.07	2.21	2.36	2.45	2.51	2.59

	11	12	13	14	15	16	17	18	19	20
RANDOM	NOT CALCULATED									
DEGREE	2.17	2.30	2.41	2.49	2.51	2.56	2.76	2.80	3.39	0.00
DVEC 3	2.36	2.37	2.38	2.51	2.57	2.68	2.79	2.91	3.26	0.00
ADJCEN	2.70	2.54	2.38	2.50	2.55	2.59	2.81	2.86	3.31	0.00
LCA	2.52	2.54	2.50	2.56	2.65	2.78	2.98	3.18	3.50	0.00

N = 20, EDGES = 95(50%)

ORDER	BRANCHES		NODES	
	avg	max	avg	max
RANDOM	NOT ENOUGH SPACE AVAIL			
DEGREE	110	1345	203	2592
DVEC 3	76	1206	143	2308
ADJCEN	46	881	93	1731
LCA	31	648	65	1281

BRANCH FACTOR BY LEVEL

	1	2	3	4	5	6	7	8	9	10
RANDOM	NOT CALCULATED									
DEGREE	1.15	1.34	1.57	1.55	1.79	1.93	1.73	1.59	1.64	1.71
DVEC 3	1.07	1.26	1.56	1.60	1.66	1.70	1.63	1.59	1.93	1.90
ADJCEN	1.00	1.00	1.00	1.14	1.68	2.00	2.11	2.37	2.25	2.52
LCA	1.00	1.00	1.00	1.14	1.65	1.85	1.92	2.15	2.27	2.32

	11	12	13	14	15	16	17	18	19	20
RANDOM	NOT CALCULATED									
DEGREE	1.89	2.02	2.01	2.38	2.41	2.54	2.66	2.79	3.09	0.00
DVEC 3	2.13	2.26	2.15	2.27	2.45	2.45	2.66	2.77	3.03	0.00
ADJCEN	2.39	2.33	2.30	2.20	2.40	2.55	2.61	2.86	3.11	0.00
LCA	2.39	2.36	2.44	2.33	2.42	2.55	2.87	3.09	3.57	0.00

N = 20, EDGES = 133(70%)

ORDER	BRANCHES avg	BRANCHES max	NODES avg	NODES max
RANDOM	NOT ENOUGH SPACE AVAIL			
DEGREE	52	182	91	329
DVEC 3	36	119	67	208
ADJCEN	21	41	43	76
LCA	19	35	41	66

BRANCH FACTOR BY LEVEL

	1	2	3	4	5	6	7	8	9	10
RANDOM	NOT CALCULATED									
DEGREE	1.02	1.03	1.07	1.17	1.45	1.45	1.67	1.64	1.88	1.87
DVEC 3	1.00	1.01	1.05	1.07	1.25	1.56	1.57	1.75	1.82	1.52
ADJCEN	1.00	1.00	1.00	1.00	1.00	1.01	1.13	1.63	1.97	2.30
LCA	1.00	1.00	1.00	1.00	1.00	1.01	1.12	1.63	1.88	2.04

	11	12	13	14	15	16	17	18	19	20
RANDOM	NOT CALCULATED									
DEGREE	1.80	1.67	2.17	2.29	2.52	2.64	2.81	3.10	3.32	0.00
DVEC 3	1.98	2.11	2.29	2.43	2.58	2.72	2.84	3.07	3.54	0.00
ADJCEN	2.42	2.52	2.64	2.80	2.65	2.76	2.99	3.13	3.47	0.00
LCA	2.23	2.41	2.53	2.65	2.77	2.98	3.19	3.37	3.91	0.00

N = 25, EDGES = 90(30%)

ORDER	BRANCHES avg	BRANCHES max	NODES avg	NODES max
RANDOM	NOT ENOUGH SPACE AVAIL			
DEGREE	NOT ENOUGH SPACE AVAIL			
DVEC 3	126	1158	251	2288
ADJCEN	63	411	132	819
LCA	33	115	73	230

BRANCH FACTOR BY LEVEL

	1	2	3	4	5	6	7	8	9	10
RANDOM	NOT CALCULATED									
DEGREE	NOT CALCULATED									
DVEC 3	1.20	1.35	1.58	1.85	1.80	1.93	1.98	2.05	1.79	1.84
ADJCEN	1.00	1.00	1.32	2.06	2.12	2.05	2.20	2.09	2.15	2.07
LCA	1.00	1.00	1.32	1.97	1.91	2.08	1.98	2.03	2.08	2.08

	11	12	13	14	15	16	17	18	19	20
RANDOM	NOT CALCULATED									

```
DEGREE    NOT CALCULATED
DVEC 3 1.77 1.85 2.00 2.07 2.05 2.15 2.32 2.47 2.37 2.51
ADJCEN 2.09 2.03 2.09 2.15 2.26 2.16 2.32 2.36 2.52 2.53
   LCA 2.02 2.03 2.09 2.22 2.35 2.42 2.48 2.54 2.61 2.75

        21   22   23   24   25
RANDOM    NOT CALCULATED
DEGREE    NOT CALCULATED
DVEC 3 2.70 2.78 2.85 3.47 0.00
ADJCEN 2.79 2.80 2.99 3.38 0.00
   LCA 2.89 3.08 3.37 3.83 0.00


N = 25, EDGES = 150(50%)

ORDER  BRANCHES      NODES
       avg  max    avg  max

RANDOM  NOT ENOUGH SPACE AVAIL
DEGREE  NOT ENOUGH SPACE AVAIL
DVEC 3 178 1328    340 2568
ADJCEN  97  932    193 1829
   LCA  38  139     76  271

                BRANCH FACTOR BY LEVEL
        1    2    3    4    5    6    7    8    9   10
RANDOM    NOT CALCULATED
DEGREE    NOT CALCULATED
DVEC 3 1.09 1.24 1.29 1.49 1.76 1.77 1.73 1.83 1.87 1.85
ADJCEN 1.00 1.00 1.00 1.12 1.40 1.93 2.18 2.26 2.24 2.11
   LCA 1.00 1.00 1.00 1.12 1.34 1.86 1.91 2.08 2.01 1.99


        11   12   13   14   15   16   17   18   19   20
RANDOM    NOT CALCULATED
DEGREE    NOT CALCULATED
DVEC 3 1.89 1.82 1.87 1.80 1.86 1.92 1.94 2.03 2.08 2.16
ADJCEN 2.07 1.98 2.02 1.92 2.04 1.97 2.06 2.01 2.12 2.25
   LCA 2.08 2.01 1.95 1.95 1.92 2.04 2.04 2.20 2.26 2.37


        21   22   23   24   25
RANDOM    NOT CALCULATED
DEGREE    NOT CALCULATED
DVEC 3 2.35 2.59 2.66 3.05 0.00
ADJCEN 2.33 2.38 2.59 3.08 0.00
   LCA 2.59 2.65 2.94 3.29 0.00
```

N = 25, EDGES = 210(70%)

ORDER	BRANCHES avg	BRANCHES max	NODES avg	NODES max
RANDOM	NOT ENOUGH SPACE AVAIL			
DEGREE	NOT ENOUGH SPACE AVAIL			
DVEC 3	NOT ENOUGH SPACE AVAIL			
ADJCEN	87	723	169	1431
LCA	34	122	64	225

BRANCH FACTOR BY LEVEL

	1	2	3	4	5	6	7	8	9	10
RANDOM	NOT CALCULATED									
DEGREE	NOT CALCULATED									
DVEC 3	NOT CALCULATED									
ADJCEN	1.00	1.00	1.00	1.00	1.00	1.01	1.15	1.48	1.90	2.06
LCA	1.00	1.00	1.00	1.00	1.00	1.01	1.15	1.43	1.78	1.88

	11	12	13	14	15	16	17	18	19	20
RANDOM	NOT CALCULATED									
DEGREE	NOT CALCULATED									
DVEC 3	NOT CALCULATED									
ADJCEN	2.22	2.34	2.22	2.16	2.14	2.13	2.14	2.13	2.05	2.08
LCA	1.94	2.05	2.12	1.95	2.05	2.02	2.08	2.04	2.10	2.27

	21	22	23	24	25
RANDOM	NOT CALCULATED				
DEGREE	NOT CALCULATED				
DVEC 3	NOT CALCULATED				
ADJCEN	2.10	2.44	2.59	2.85	0.00
LCA	2.36	2.58	2.84	3.41	0.00

N = 30, EDGES = 130(30%)

ORDER	BRANCHES avg	BRANCHES max	NODES avg	NODES max
RANDOM	NOT ENOUGH SPACE AVAIL			
DEGREE	198	1504	389	2922
DVEC 3	88	412	179	815
ADJCEN	63	691	134	1391
LCA	31	57	71	120

BRANCH FACTOR BY LEVEL

	1	2	3	4	5	6	7	8	9	10

```
RANDOM   NOT CALCULATED
DEGREE 1.36 1.73 1.97 1.82 1.89 1.97 1.92 1.86 1.94 1.93
DVEC 3 1.21 1.48 1.64 1.74 1.94 1.91 1.92 1.95 1.92 2.03
ADJCEN 1.00 1.00 1.30 1.86 2.16 2.21 2.26 2.22 2.32 2.25
   LCA 1.00 1.00 1.30 1.83 1.94 2.04 2.10 2.13 2.13 2.16


       11   12   13   14   15   16   17   18   19   20
RANDOM   NOT CALCULATED
DEGREE 2.09 2.18 2.08 1.97 2.05 2.04 2.12 2.15 2.37 2.32
DVEC 3 1.95 2.05 2.16 2.33 2.25 2.22 2.24 2.36 2.30 2.34
ADJCEN 2.28 2.21 2.25 2.20 2.31 2.32 2.21 2.16 2.31 2.39
   LCA 2.19 2.14 2.23 2.19 2.21 2.21 2.24 2.32 2.36 2.38


       21   22   23   24   25   26   27   28   29   30
RANDOM   NOT CALCULATED
DEGREE 2.26 2.39 2.43 2.42 2.51 2.42 2.59 2.70 2.80 0.00
DVEC 3 2.41 2.32 2.54 2.53 2.62 2.54 2.59 2.63 2.84 0.00
ADJCEN 2.41 2.42 2.42 2.58 2.51 2.50 2.55 2.79 2.87 0.00
   LCA 2.46 2.51 2.52 2.60 2.72 2.81 3.03 3.12 3.41 0.00


N = 30, EDGES = 217(50%)

ORDER  BRANCHES      NODES
       avg  max    avg  max

RANDOM  NOT ENOUGH SPACE AVAIL
DEGREE  NOT ENOUGH SPACE AVAIL
DVEC 3 289 1825   555 3652
ADJCEN  79 1995   163 3959
   LCA  32   76    70  148

                BRANCH FACTOR BY LEVEL
        1    2    3    4    5    6    7    8    9   10
RANDOM   NOT CALCULATED
DEGREE   NOT CALCULATED
DVEC 3 1.08 1.12 1.33 1.59 1.70 1.91 1.89 2.13 1.99 1.82
ADJCEN 1.00 1.00 1.00 1.01 1.13 1.64 2.01 2.20 2.25 2.33
   LCA 1.00 1.00 1.00 1.01 1.12 1.63 1.88 2.00 2.01 2.08


       11   12   13   14   15   16   17   18   19   20
RANDOM   NOT CALCULATED
DEGREE   NOT CALCULATED
DVEC 3 1.82 1.80 1.76 1.94 1.89 2.04 2.12 2.18 2.17 2.23
ADJCEN 2.30 2.29 2.33 2.27 2.28 2.32 2.34 2.13 2.28 2.39
```

```
   LCA 2.18 2.09 2.13 2.20 2.15 2.18 2.29 2.24 2.34 2.40

       21   22   23   24   25   26   27   28   29   30
RANDOM   NOT CALCULATED
DEGREE   NOT CALCULATED
DVEC 3 2.33 2.43 2.44 2.51 2.54 2.62 2.83 2.94 3.17 0.00
ADJCEN 2.40 2.37 2.43 2.45 2.58 2.62 2.78 2.96 3.25 0.00
   LCA 2.50 2.55 2.66 2.64 2.80 2.92 3.06 3.31 3.75 0.00


N = 30, EDGES = 304(70%)

ORDER  BRANCHES      NODES
       avg  max    avg  max

RANDOM  NOT ENOUGH SPACE AVAIL
DEGREE  NOT ENOUGH SPACE AVAIL
DVEC 3 215 1519    394 2896
ADJCEN  46 1003     96 2003
   LCA  30   67     65  132

                BRANCH FACTOR BY LEVEL
        1    2    3    4    5    6    7    8    9   10
RANDOM   NOT CALCULATED
DEGREE   NOT CALCULATED
DVEC 3 1.00 1.00 1.04 1.05 1.08 1.22 1.36 1.34 1.57 1.65
ADJCEN 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.05 1.28
   LCA 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.05 1.28

       11   12   13   14   15   16   17   18   19   20
RANDOM   NOT CALCULATED
DEGREE   NOT CALCULATED
DVEC 3 1.97 2.03 2.04 1.99 1.83 1.72 1.68 2.07 2.16 2.24
ADJCEN 1.67 1.97 2.19 2.35 2.48 2.52 2.52 2.45 2.58 2.55
   LCA 1.66 1.88 1.96 2.09 2.10 2.26 2.34 2.34 2.45 2.45

       21   22   23   24   25   26   27   28   29   30
RANDOM   NOT CALCULATED
DEGREE   NOT CALCULATED
DVEC 3 2.33 2.49 2.64 2.70 2.84 2.98 3.15 3.34 3.75 0.00
ADJCEN 2.60 2.69 2.89 2.81 2.93 3.04 3.18 3.39 3.83 0.00
   LCA 2.68 2.70 2.86 2.98 3.13 3.28 3.48 3.75 4.23 0.00


  * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                OUTPUT FROM GRAPH DIVIDING ROUTINE
  * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

N	EDGES	AVG # SUBS	AVG SIZE LARGEST SUB	
			n	edges
6				
	6	3.4	3.6	3.6
	7	2.5	4.4	5.2
	8	1.7	5.3	7.1
	9	1.3	5.7	8.7
	10	1.1	5.9	9.9
	11	1.0	6.0	11.0
	12	1.0	6.0	12.0
	13	1.0	6.0	13.0
8				
	8	5.4	3.5	3.5
	10	2.6	6.1	7.6
	12	1.4	7.5	11.3
	14	1.1	7.9	13.9
	16	1.0	8.0	15.9
	18	1.0	8.0	18.0
	20	1.0	8.0	20.0
	22	1.0	8.0	22.0
10				
	10	7.0	4.0	4.0
	12	4.2	6.3	7.7
	14	2.7	8.2	12.0
	16	1.9	9.0	14.8
	18	1.4	9.5	17.3
	20	1.2	9.7	19.5
	22	1.1	9.9	21.8
	24	1.0	10.0	24.0
	26	1.0	10.0	26.0
	28	1.0	10.0	28.0
12				
	12	8.8	4.2	4.2
	15	4.3	8.1	10.4
	18	2.5	10.2	15.5
	21	1.7	11.3	20.0
	24	1.4	11.6	23.3
	27	1.1	11.9	26.8
	30	1.0	12.0	30.0
	33	1.0	12.0	33.0
	36	1.0	12.0	36.0
	39	1.0	12.0	39.0
14				
	14	10.4	4.6	4.6
	17	6.0	8.1	10.3
	20	3.5	11.1	16.4
	23	2.4	12.5	21.1
	26	1.8	13.1	24.8

	29	1.5	13.5 28.2
	32	1.2	13.8 31.5
	35	1.2	13.8 34.6
	38	1.1	13.9 37.9
	41	1.0	14.0 41.0
	44	1.0	14.0 44.0
	47	1.0	14.0 47.0
16			
	16	12.9	4.1 4.1
	20	5.3	10.7 13.8
	24	1.5	15.2 22.8
	28	1.1	15.8 27.7
	32	1.1	15.9 31.9
	36	1.0	15.9 36.0
	40	1.0	16.0 40.0
	44	1.0	16.0 44.0
18			
	18	14.1	4.9 4.9
	22	7.4	10.5 13.4
	26	4.4	13.9 20.9
	30	2.8	15.9 27.2
	34	2.1	16.9 32.7
	38	1.6	17.2 36.7
	42	1.4	17.6 41.3
	46	1.2	17.7 45.5
	50	1.1	17.9 49.9
	54	1.0	17.9 53.8
	58	1.0	18.0 58.0
	62	1.0	18.0 62.0
	66	1.0	18.0 66.0
	70	1.0	18.0 69.9
20			
	20	16.4	4.6 4.6
	25	7.5	12.3 16.0
	30	4.1	16.2 24.7
	35	2.4	18.5 32.6
	40	1.8	19.1 38.8
	45	1.4	19.6 44.3
	50	1.2	19.8 49.8
	55	1.1	19.8 54.8
	60	1.0	20.0 59.8
	65	1.0	20.0 65.0
	70	1.0	20.0 69.9
	75	1.0	20.0 75.0
	80	1.0	20.0 80.0
	85	1.0	20.0 85.0
22			
	22	18.0	5.0 5.0

27	8.9	12.9	16.8
32	5.5	16.5	24.7
37	3.5	18.9	32.6
42	2.3	20.5	40.0
47	1.7	21.1	45.5
52	1.5	21.4	50.6
57	1.3	21.7	56.1
62	1.2	21.8	61.5
67	1.2	21.8	66.4
72	1.1	21.9	71.8
77	1.0	22.0	77.0
82	1.0	22.0	82.0
87	1.0	22.0	87.0
92	1.0	22.0	92.0
24			
24	19.8	5.2	5.2
30	8.4	15.2	19.7
36	4.5	19.7	30.1
42	2.6	22.1	39.3
48	1.7	23.1	46.4
54	1.3	23.7	53.4
60	1.2	23.8	59.8
66	1.1	23.9	65.8
72	1.0	24.0	72.0
78	1.0	24.0	78.0
84	1.0	24.0	84.0
90	1.0	24.0	90.0
26			
26	21.8	5.2	5.2
32	10.5	15.1	19.8
38	6.1	19.9	30.0
44	3.9	22.3	38.5
50	2.7	23.8	46.5
56	2.1	24.7	53.6
62	1.6	25.2	60.6
68	1.3	25.6	67.1
74	1.2	25.7	73.0
80	1.1	25.9	79.7
86	1.0	26.0	85.9
92	1.0	25.9	91.6
98	1.0	26.0	98.0
104	1.0	26.0	104.0
110	1.0	26.0	110.0
116	1.0	26.0	116.0
28			
28	23.6	5.4	5.4
35	10.5	17.0	22.4
42	5.5	22.8	35.5

49	3.4	25.3	45.5
56	2.2	26.6	54.1
63	1.6	27.2	61.3
70	1.3	27.6	69.1
77	1.2	27.7	76.3
84	1.1	27.9	83.8
91	1.1	27.9	90.9
98	1.0	28.0	98.0
105	1.0	28.0	104.8
112	1.0	28.0	112.0
119	1.0	28.0	119.0

30

30	25.5	5.6	5.6
37	11.9	17.6	22.7
44	6.7	23.5	36.0
51	4.4	25.7	44.8
58	2.7	28.1	55.6
65	2.0	28.7	63.0
72	1.5	29.3	70.3
79	1.5	29.5	78.2
86	1.3	29.7	85.5
93	1.1	29.9	92.8
100	1.0	29.9	99.7
107	1.0	30.0	107.0
114	1.0	30.0	114.0
121	1.0	30.0	121.0
128	1.0	30.0	128.0
135	1.0	30.0	135.0

40

40	34.8	6.2	6.2
60	7.9	31.7	49.0
80	2.7	37.8	75.9
100	1.5	39.4	98.6
120	1.1	39.9	119.8
140	1.0	40.0	140.0
160	1.0	40.0	160.0
180	1.0	40.0	180.0

50

50	44.5	6.5	6.5
75	10.6	38.7	61.2
100	4.0	46.7	95.6
125	2.0	48.5	120.8
150	1.4	49.5	148.7
175	1.1	49.9	174.9
200	1.0	50.0	200.0
225	1.0	50.0	225.0
250	1.0	50.0	250.0

60

	60	54.3	6.7 6.7
	90	12.7	46.1 72.2
	120	4.5	56.0 113.9
	150	2.1	58.7 147.5
	180	1.2	59.7 179.0
	210	1.1	59.9 209.6
	240	1.0	60.0 240.0
	270	1.0	60.0 270.0
	300	1.0	60.0 300.0
	330	1.0	60.0 330.0
70			
	70	64.3	6.7 6.7
	105	15.1	53.7 85.0
	140	5.7	64.6 132.1
	175	2.6	68.2 171.9
	210	1.5	69.3 207.8
	245	1.2	69.8 244.8
	280	1.0	70.0 280.0
	315	1.0	70.0 315.0
	350	1.0	70.0 350.0
80			
	80	74.3	6.7 6.7
	120	14.6	63.9 99.8
	160	4.7	75.8 153.6
	200	2.0	78.7 197.7
	240	1.3	79.7 239.6
	280	1.0	80.0 280.0
	320	1.0	80.0 320.0
	360	1.0	80.0 360.0
	400	1.0	80.0 400.0
90			
	90	83.8	7.2 7.2
	135	19.1	68.6 107.8
	180	6.5	83.4 169.9
	225	3.1	87.6 220.8
	270	1.9	89.0 267.4
	315	1.2	89.8 314.5
	360	1.0	89.9 360.0
	405	1.0	90.0 405.0
	450	1.0	90.0 450.0
	495	1.0	90.0 495.0
100			
	100	93.0	8.0 8.0
	150	20.9	77.1 122.5
	200	7.2	91.9 186.0
	250	3.0	97.8 246.4
	300	1.6	99.4 299.1
	350	1.2	99.7 349.0

```
       400          1.1                  99.9 399.2
       450          1.0                 100.0 450.0
       500          1.0                 100.0 500.0
       550          1.0                 100.0 550.0


 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                  OUTPUT FROM DIVIDE AND COLOR PROGRAM
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

N = 25, EDGES =   6( 2%)

ORDER  BRANCHES      NODES
       avg   max    avg   max
RANDOM   6     6     12    12
DEGREE   6     6     12    12
DEG  3   6     6     12    12
ADJCEN   6     6     12    12
   LCA   6     6     12    12


N = 25, EDGES =  12( 4%)

ORDER  BRANCHES      NODES
       avg   max    avg   max
RANDOM  12    12     23    24
DEGREE  12    12     23    24
DEG  3  12    12     23    24
ADJCEN  12    12     23    24
   LCA  12    12     23    24


N = 25, EDGES =  18( 6%)

ORDER  BRANCHES      NODES
       avg   max    avg   max
RANDOM  17    18     34    36
DEGREE  17    19     34    36
DEG  3  17    18     34    36
ADJCEN  17    18     34    36
   LCA  17    18     34    36


N = 25, EDGES =  24( 8%)

ORDER  BRANCHES      NODES
       avg   max    avg   max
RANDOM  21    25     43    55
```

DEGREE	22	46	46	90
DEG 3	21	27	44	55
ADJCEN	21	23	43	54
LCA	21	23	44	54

N = 25, EDGES = 30(10%)

ORDER	BRANCHES		NODES	
	avg	max	avg	max
RANDOM	25	45	53	91
DEGREE	27	52	57	107
DEG 3	24	31	52	68
ADJCEN	23	30	52	67
LCA	23	30	52	68

N = 25, EDGES = 36(12%)

ORDER	BRANCHES		NODES	
	avg	max	avg	max
RANDOM	51	945	108	1893
DEGREE	34	191	72	380
DEG 3	26	94	59	189
ADJCEN	24	35	57	82
LCA	24	35	57	82

N = 25, EDGES = 42(14%)

ORDER	BRANCHES		NODES	
	avg	max	avg	max
RANDOM	84	1012	172	2021
DEGREE	43	208	91	425
DEG 3	29	86	64	177
ADJCEN	29	170	65	349
LCA	25	54	58	109

* *

RESULTS OF VARYING EDGES FOR A CONSTANT N

* *

N = 20, EDGES = 38(20%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes

DEGREE	46	363	96	731	5.89	6.59
DVEC 3	27	137	60	278	4.92	5.63
ADJCEN	23	82	52	166	4.41	5.11
LCA	20	33	46	70	3.50	4.25

N = 20, EDGES = 47(25%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	45	127	94	248	4.84	5.51
DVEC 3	28	93	62	182	4.53	5.20
ADJCEN	21	43	52	87	3.76	4.47
LCA	20	27	48	63	3.30	4.14

N = 20, EDGES = 57(30%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	49	187	99	380	5.23	5.94
DVEC 3	32	164	67	320	5.10	5.77
ADJCEN	25	109	56	223	4.69	5.41
LCA	21	55	48	117	4.01	4.76

N = 20, EDGES = 66(35%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	112	1119	219	2198	7.02	7.70
DVEC 3	52	308	104	608	5.73	6.41
ADJCEN	38	513	80	1015	6.24	6.92
LCA	23	62	51	123	4.13	4.81

N = 20, EDGES = 76(40%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	NOT ENOUGH SPACE AVAIL					
DVEC 3	59	267	116	524	5.59	6.26
ADJCEN	30	292	65	593	5.68	6.39
LCA	23	285	52	568	5.65	6.34

N = 20, EDGES = 85(45%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	92	635	173	1206	6.45	7.10
DVEC 3	56	425	109	818	6.05	6.71
ADJCEN	37	379	77	749	5.94	6.62
LCA	23	64	49	128	4.16	4.85

N = 20, EDGES = 95(50%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	101	811	188	1608	6.70	7.38
DVEC 3	57	367	107	676	5.91	6.52
ADJCEN	34	501	71	980	6.22	6.89
LCA	28	584	60	1150	6.37	7.05

N = 20, EDGES = 104(55%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	106	431	194	777	6.07	6.66
DVEC 3	69	440	131	844	6.09	6.74
ADJCEN	34	258	70	497	5.55	6.21
LCA	23	79	48	156	4.37	5.05

N = 20, EDGES = 114(60%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	99	969	180	1696	6.88	7.44
DVEC 3	62	576	114	1029	6.36	6.94
ADJCEN	37	561	75	1088	6.33	6.99
LCA	27	438	56	855	6.08	6.75

N = 20, EDGES = 123(65%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	86	765	155	1351	6.64	7.21
DVEC 3	55	468	101	832	6.15	6.72
ADJCEN	28	188	57	365	5.24	5.90
LCA	21	53	43	103	3.97	4.63

N = 20, EDGES = 133(70%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	74	419	130	748	6.04	6.62
DVEC 3	47	282	86	498	5.64	6.21
ADJCEN	21	57	44	114	4.04	4.74
LCA	20	33	41	62	3.50	4.13

N = 20, EDGES = 142(75%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	60	295	103	499	5.69	6.21
DVEC 3	38	226	69	403	5.42	6.00
ADJCEN	22	66	43	118	4.19	4.77
LCA	19	37	39	71	3.61	4.26

N = 20, EDGES = 152(80%)

order	branches		nodes		Logarithm of max	
	avg	max	avg	max	branches	nodes
DEGREE	45	143	76	237	4.96	5.47
DVEC 3	30	71	54	118	4.26	4.77
ADJCEN	20	46	39	81	3.83	4.39
LCA	19	27	37	49	3.30	3.89

LIST OF REFERENCES

1. Dailey, David P.[1978]. "Graph Coloring by Humans and Machines: A Polynomial Complete Problem Solving Task," Doctoral Thesis, University of Colorado, Boulder, Colo.

2. Johnson, David S.[1973]. "Approximation Algorithms for Combinatorial Problems," Proceedings of 5th Annual ACM Symp. on the Theory of Computing, 38-49.

3. Matula, D.W., Marble, G., and Isaacson, J.D.[1972]. "Graph Coloring Algorithms," Graph Theory and Computing, R.C. Read (ed), Academic Press, N.Y.

4. Williams, M.R.[1974]. "Heuristic Proceedures (If They Work - Leave Them Alone)," Software - Practice and Experience 4, 237-240.

5. Garey, M.R. and Johnson, D.S.[1979]. Computers and Intractability, A Guide to the Theory of NP-Completeness, W.H. Freeman, San Francisco, Ca.

6. Baase, Sara[1978]. Computer Algorithms, Introduction to Design and Analysis, Addison-Wesley, Reading, Mass.

7. Cook, S.A.[1971]. "The Complexity of Theorem-Proving Proceedures," Proc. 3rd Annual ACM Symp. on Theory of Computing, ACM, N.Y. 151-158.

8. Karp, R.M.[1972]. "Reducibility among Combinatorial Problems," R.E. Miller and J.W. Thatcher (eds), Complexity of Computer Computations, Plenum Press, N.Y., 85-103.

9. Grogono, Peter[1978]. Programming in PASCAL, Addison-Wesley, Reading, Mass.

10. Tremblay, J. and Bunt, R.B.[1979]. An Introduction to Computer Science: An Algorithmic Approach, McGraw-Hill, N.Y.

11. Welsh, D.J.A. and Powell, M.B.[1967]. "An Upper Bound for the Chromatic Number of a Graph and its Application to Timetabling Problems," Computer Journal, 10,1,85-86.

12. Liu, C.L.[1968]. Introduction to Combinatorial Mathematics, McGraw-Hill, N.Y.

13. Tomescu, Ioan[1975]. Introduction to Combinatorics, Collet's, London, Eng.

14. Garey, M.R. and Johnson, D.S.[1976]. "The Complexity of near Optimal Graph Coloring," Journal ACM, 23, 43-49.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Asst Professor Douglas R. Smith, Code 52SC Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	LCDR Ronald E. Rautenberg, USNR 10610 235th St S.W. Edmonds, Washington 98020	2

Thesis 191192
R24525 Rautenberg
c.1 An investigation of
several branching
functions in a branch
and bound algorithm
for the chromatic
number problem.